

Ejercicios-Juez-resueltos.pdf



beaas_218



Fundamentos de Algoritmia



2º Grado en Ingeniería del Software



**Facultad de Informática
Universidad Complutense de Madrid**

EJERCICIOS RESUELTOS FAL

ÍNDICE

Duplicar la entrada a)	3
Duplicar la entrada b)	4
Número de máximos	5
¿Está bien dividido el vector?	7
Temperaturas extremas	9
Pasando revista	11
Un móvil nuevo	13
Rescate aéreo 1	16
¿Cómo van las encuestas?	18
Reservas de elefantes	20
Los hermanos Dalton	23
Avitualamiento en las etapas ciclistas	24
Viajes a Marte	27
Cintas de colores	29
Releyendo un libro. Versión 1	32
Releyendo un libro	35
Invertir un número	37
Dígitos complementarios	39
Vector parcialmente ordenado	41
Números caucásicos	43
Sumar dos números dígito a dígito	45
Móviles	48
Fuga de la prisión	50
El juego del bongo	52
Buscar el elemento perdido	54
Obtener el mínimo de un vector	56
Degrado de una imagen	58
Battlestar Galactica	60
Las N reinas	63
Compra de la semana	65
Los funcionarios del Ministerio	69
Noticias de Inglaterra (torres y reinas)	72
Papa Noel reparte juguetes (simplificado)	75

Papa Noel reparte juguetes (optimización)	77
Adornando la casa por Navidad.....	80
Caminando voy	84
Todos con la selección.....	86
Transformar un número decimal a binario	88
Números interesantes.....	91
Elementos bien situados	92
Impares borrados	94
Papa Noel reparte juguetes.....	96
Concierto benéfico	99

EJERCICIOS

Duplicar la entrada a)

Enunciado:

Descripción

Crea un programa que duplique los números enteros leídos por entrada estándar.

Entrada

La entrada comenzará con una linea conteniendo un número natural N que indica la cantidad de números que habrá que duplicar. A continuación le siguen N líneas, cada una con un número entero (puede ser positivo o negativo) que habrá que duplicar.

Salida

Por cada número considerado se mostrará en la salida estándar una línea con el doble del número leído.

Ejemplo de entrada

```
3
1
3
12
```

Ejemplo de salida

```
2
6
24
```

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

/*
 *  ESPECIFICACIÓN FORMAL
 *  -----
 *  P = {   }
 *  fun resolver(n : ent) dev sol : ent
 *  Q = { sol = 2 * n }
 *
 */
// función que resuelve el problema
int resolver(int n) {
    return 2 * n; // coste O(1)
}
```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;

    int sol = resolver(n);

    // escribir sol
    cout << sol << '\n';
}


```

Duplicar la entrada b)

Enunciado:

Descripción

Crea un programa que duplique los números enteros leídos por entrada estándar.

Entrada

La entrada consistirá en un número de líneas, cada una conteniendo un número entero que se debe duplicar. La entrada termina cuando el número leído sea un 0, caso que no se debe procesar.

Salida

Por cada número considerado se mostrará en la salida estándar una línea con el doble del número leído.

Ejemplo de entrada

```

1
3
12
0

```

Ejemplo de salida

```

2
6
24

```

Solución:

```

// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

```

```

/*
 *      ESPECIFICACIÓN FORMAL
 *
 * -----
 *      P = {   }
 *      fun resolver(n : ent) dev sol : ent
 *      Q = { sol = 2 * n }
 *
 */

// función que resuelve el problema
int resolver(int n) {
    return 2 * n; // coste O(1)
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    if (n == 0)
        return false;

    int sol = resolver(n);

    // escribir sol
    cout << sol << '\n';

    return true;
}

```

Número de máximos

Enunciado:

Hemos desarrollado un juego online y tenemos en un fichero las puntuaciones de diversos jugadores. Ahora queremos saber cual ha sido la puntuación máxima obtenida y cuantos jugadores la han obtenido. Ha de tenerse en cuenta que el juego permite tener puntuaciones negativas cuando el jugador pierde más puntos de los que obtiene, pero ningún jugador termina con cero puntos.

El problema es que juego ha tenido tanto éxito que no es posible cargar todo el fichero en memoria, por lo tanto tendremos que resolver el problema al tiempo que vamos leyendo el fichero.



Requisitos de implementación.

No se pueden almacenar los valores de entrada en memoria.

Entrada

La entrada comienza con un valor entero que indica el número de casos de prueba. Cada caso de prueba consta de una línea con las puntuaciones obtenidas por los jugadores. Cada caso termina con el valor 0.

No puede suponerse un límite en el número de puntuaciones obtenidas por los jugadores. Las puntuaciones son números enteros diferentes de cero y se sabe que pueden almacenarse en una variable de tipo int.

Salida

Para cada caso de prueba se escribe en una línea el valor máximo y el número de veces que se repite separados por un carácter blanco.

Entrada de ejemplo

```
3
5 7 3 4 7 3 2 6 5 4 7 1 2 7 3 0
6 6 6 0
5 0
```

Salida de ejemplo

```
7 4
6 3
5 1
```

Autor: Isabel Pita

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

/*
 * ACLARACIONES:
 * N = número de puntuaciones
 * v = supuesto vector que representa las posiciones (aunque en la implementación no se almacenen los valores)
 *
 * ESPECIFICACIÓN FORMAL
 *
 * -----
 * P = { ParaTodo p : 0 <= p < N : v[p] != 0 }
 * fun resolver() dev mejorMax : ent, numMax : ent
 * Q = { mejorMax = max i : 0 <= i < N : V[i] & numMax = #i : 0 <= i < N : V[i] = mejorMax }
 *
 */

// función que resuelve el problema
/* La función es de coste lineal O(N), donde N es el número de puntuaciones, ya que la instrucción
 * crítica que determina el coste asintótico del programa se ejecuta N veces */
pair<int, int> resolver() {

    int aux; cin >> aux;
    int mejorMax = aux, numMax = 1;
    cin >> aux;
    while (aux != 0) { // coste O(N), donde N es el número de puntuaciones (da N vueltas)
        // todas las operaciones interiores del bucle son de coste O(1)
        if (aux > mejorMax) {
            mejorMax = aux;
            numMax = 1;
        }
        else if (aux == mejorMax)
            ++numMax;
        cin >> aux; // podemos tomar esta instrucción como instrucción crítica
    }
    return { mejorMax, numMax };
}
```

```
// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {
    pair<int, int> sol = resolver();
    // escribir sol:
    cout << sol.first << ' ' << sol.second << '\n';
}
```

¿Está bien dividido el vector?

Enunciado:

Nos dan un vector de números enteros con n elementos y una cierta posición p del vector. Debemos comprobar si todas las componentes del vector en las posiciones $[0..p]$ (incluida la p) son menores (estrictamente) que todas las componentes en las posiciones $[p+1..n-1]$.

Si $p=n-1$ el resultado debe ser cierto.

Requisitos de implementación.

Se implementará una función que reciba un vector con los valores y el valor p y que devuelva un valor booleano que indique si el vector está bien dividido.

El orden de complejidad del algoritmo debe ser lineal respecto al número de elementos del vector.

Entrada

La entrada comienza con un valor entero que indica el número de casos de prueba. Cada caso de prueba consta de dos líneas. La primera indica el número de elementos del vector y la posición que debería separar los valores menores de los mayores. La segunda línea contiene los valores del vector.

El número de valores de entrada n cumple: $1 \leq n \leq 300.000$ y la posición p cumple $0 \leq p \leq n-1$. Los valores del vector son números enteros que pueden almacenarse en una variable de tipo `int`.

Salida

Para cada caso de prueba se escribe en una línea *SI* si la posición dada separa el vector en la forma apropiada y *NO* si no lo hace.

Entrada de ejemplo

```
5
15 5
5 7 3 4 7 3 8 9 8 10 8 9 8 9 10
4 0
4 7 5 6
4 0
8 7 5 6
4 3
4 3 6 2
5 2
5 3 4 6 4
```

Salida de ejemplo

```
SI
SI
NO
SI
NO
```

Autor: Isabel Pita

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ACLARACIONES:
 *   N = vct.size()
 *
 *   ESPECIFICACIÓN FORMAL
 *   -----
 *   P = { 1 <= N <= 300.000 & 0 <= p <= n - 1 }
 *   fun resolver(vct[0..N] de ent, N : ent, p : ent) dev sol : bool
 *   Q = { sol = ParaTodo i,j : 0 <= i <= p < j < N : vct[i] < vct[j] }
 *
 */

// función que resuelve el problema
/* El coste asintótico de esta función será del orden O(N), donde N es el número de elementos del vector.
   Esto lo podemos ver fácilmente dándonos cuenta de que, obviando las instrucciones que tienen coste O(1),
   los dos bucles <for> que quedan dan en total p + (N - p) = N vueltas (y las instrucciones del interior de
   los bucles son a su vez de coste constante */
bool resolver(vector<int> const& vct, int N, int p) {

    if (p == N - 1) // coste O(1)
        return true;
    int mejorMaxIzq = vct[0]; // coste O(1)
    for (int i = 0; i <= p; ++i) // p vueltas
        if (vct[i] > mejorMaxIzq) // coste O(1)
            mejorMaxIzq = vct[i];
    int mejorMinDch = vct[p + 1];
    for (int j = p + 1; j < N; ++j) // N - p vueltas
        if (vct[j] < mejorMinDch) // coste O(1)
            mejorMinDch = vct[j];
    if (mejorMaxIzq < mejorMinDch) // coste O(1)
        return true;
    else
        return false;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    int N; cin >> N;
    int p; cin >> p;
    vector<int> vct(N);
    for (int i = 0; i < N; ++i)
        cin >> vct[i];

    bool sol = resolver(vct, N, p);

    // escribir sol
    if (sol) cout << "SI";
    else cout << "NO";
    cout << '\n';
}
```

Temperaturas extremas

Enunciado:

Doc sigue muy preocupado por las temperaturas que alcanza su DeLorean durante los viajes en el tiempo. Para mayor precisión, ha registrado la temperatura del coche durante cada milisegundo de cada viaje, medida en milmillonésimas de grados Kelvin. Y es hora de procesar todos esos valores para saber cuál ha sido la temperatura mínima y máxima leída, así como cuántos milisegundos en total han tenido ambas lecturas.



Entrada

La entrada comienza con un valor entero que indica el número de casos de prueba que aparecen a continuación.

Cada caso de prueba consta de una única línea y contiene las distintas lecturas del termostato del coche (como mucho 10.000) terminadas con un 0. Se garantiza que hay al menos una lectura y que todas ellas son números positivos mayores que cero y menores que 10^{18} .

Salida

Para cada caso de prueba se escribirá una única línea con cuatro números separados por un espacio. Los dos primeros indican la temperatura más baja leída y cuántas veces ha aparecido. Los dos últimos hacen lo mismo pero con la temperatura más alta.

Entrada de ejemplo

```
3
1 2 3 4 0
3 3 3 3 0
9876543200 9876543210 9876543200 0
```

Salida de ejemplo

```
1 1 4 1
3 4 3 4
9876543200 2 9876543210 1
```

Notas

1. Especifica (*precondición* y *postcondición*) una función que calcule los datos solicitados para un vector de temperaturas.
2. Deriva formalmente el algoritmo correspondiente. Incluye como comentarios en el código el *invariante* y la *función de cota* del bucle.
3. El *orden de complejidad* del tiempo de ejecución del algoritmo debe ser *lineal* respecto al tamaño del vector de temperaturas..

Solución:

```
✉// Beatriz Espinar Aragón
└─// F19
```

```

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 * ACLARACIONES:
 * - N = temperaturas.size()
 *
 * ESPECIFICACIÓN FORMAL
 * -----
 * P = { 0 < N <= 10.000 } \ ParaTodo w : 0 <= w < N : 0 < temperaturas[w] < 10^18 }
 * fun resolver(temperaturas[0..N) de ent) dev <min : ent, cuantos_min : ent, max : ent, cuantos_max : ent>
 * Q = { min = minimo(temperaturas, N) /\ cuantos_min = # w : 0 <= w < N : temperaturas[w] = min /\ max = maximo(temperaturas, N) /\ 
 *       cuantos_max = # u : 0 <= u < N : temperaturas[u] = max }
 *
 * donde se utilizan los siguientes predicados auxiliares:
 * - minimo(v, n) = (min w : 0 <= w < n : v[w])
 * - maximo(v, n) = (max w : 0 <= w < n : v[w])
 *
 * INVARIANTE
 * -----
 * I = { 0 <= cont <= N /\ min = minimo(temperaturas, cont) /\ cuantos_min = # w : 0 <= w < cont : temperaturas[w] = min /\ 
 *       max = maximo(temperaturas, cont) /\ cuantos_max = # u : 0 <= u < cont : temperaturas[u] = max }
 * b = cont < N
 *
 * FUNCIÓN DE COTA
 * -----
 * C = N - cont
 * - C >= 0 al empezar el bucle (N - cont = N - 0 = N > 0 por la precondition)
 * - C decrece al avanzar el bucle (cont aumenta --> N - cont disminuye)
 *
 * COSTE
 * -----
 * El coste asintótico en tiempo de la función es del orden O(N), dado que recorremos las N posiciones del vector
 * pasando por cada elemento 1 vez y realizando operaciones de coste O(1) sobre ellos.
 */

```

```

struct solucion {
    long long int min;
    int cuantos_min;
    long long int max;
    int cuantos_max;
};

// función que resuelve el problema
solucion resolver(vector<long long int> const& temperaturas) {

    // inicializamos:
    long long int mejor_min = -1, mejor_max = -1;
    int cuantos_min = 0, cuantos_max = 0;
    int cont = 0;

    // bucle:
    while (cont < temperaturas.size()) {

        // comprobamos mínimo:
        if (mejor_min == -1 || temperaturas[cont] < mejor_min) {
            mejor_min = temperaturas[cont];
            cuantos_min = 1;
        }
        else if (temperaturas[cont] == mejor_min)
            ++cuantos_min;

        // comprobamos máximo:
        if (mejor_max == -1 || temperaturas[cont] > mejor_max) {
            mejor_max = temperaturas[cont];
            cuantos_max = 1;
        }
        else if (temperaturas[cont] == mejor_max)
            ++cuantos_max;

        ++cont;
    }

    return { mejor_min, cuantos_min, mejor_max, cuantos_max };
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {
    // leer los datos de la entrada
    vector<long long int> temperaturas;
    long long int aux; cin >> aux;
    while (aux != 0) {
        temperaturas.push_back(aux);
        cin >> aux;
    }

    solucion sol = resolver(temperaturas);

    // escribir sol
    cout << sol.min << " " << sol.cuantos_min << " " << sol.max << " " << sol.cuantos_max << '\n';
}

```

Pasando revista

Enunciado:

El rey de Falfanistán pasa revista a sus caballeros cada domingo por la mañana. Estos se colocan en una fila sobre sus caballos y el rey pasa por delante saludando. Como está aburrido de esta rutina, mientras pasa revista hace diversas cábaldas. En concreto, como cada caballero (subido sobre su caballo) tiene una altura diferente, el rey se pregunta cuántos caballeros habrá seguidos sin que haya tres seguidos de altura estrictamente creciente.



Vamos a suponer que las alturas de los caballeros vienen dadas en una secuencia de enteros.

1. *Especifica formalmente* una función que, dado un vector de enteros de longitud ≥ 0 , devuelva la longitud del segmento más largo que no contiene ningún subsegmento estrictamente creciente de tres elementos.
2. *Diseña e implementa* un algoritmo iterativo que resuelva el problema propuesto.
3. Escribe el *invariante* del bucle que permite demostrar la corrección del mismo y proporciona una *función de cota*.
4. Indica el *coste asintótico* del algoritmo en el caso peor y justifica adecuadamente tu respuesta.

Entrada

La entrada comienza con una línea que contiene el número de casos de prueba. Cada caso de prueba contendrá el valor del número de elementos n y a continuación los elementos de la secuencia.

Salida

Por cada caso de prueba el programa escribirá una línea con la longitud del segmento más largo solicitado en el enunciado.

Entrada de ejemplo

```
5
8
1 2 3 1 2 0 4 5
2
2 -1
5
1 1 1 2 2
6
0 4 1 2 -3 7
1
7
```

Salida de ejemplo

```
6
2
5
6
1
```

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

/*
 *   ACLARACIONES:
 *   - N = alturas.size()
 *
 *   ESPECIFICACIÓN FORMAL
 *   -----
 *   P = { N >= 0 }
 *   fun resolver(alturas[0..N) de ent) dev max_long : ent
 *   Q = { max_long = max ini, fin : 0 <= ini < fin <= N && intervalo_valido(alturas, ini, fin) : fin - ini }
 *
 *   donde se utilizan los siguientes predicados auxiliares:
 *   - intervalo_valido(v, i, f) = (!Existe u, w : i <= u < w <= f && w - u = MAX_SEGUIDOS : creciente(v, u, w))
 *   - creciente(v, i, j) = (ParaTodo w : i <= w < j - 1 : v[w] < v[w + 1])
 *
 *   INVARIANTE
 *   -----
 *   I = { 1 <= fin <= N /\ 0 <= ini < fin /\ max_long = max i, j : 0 <= i < j <= fin && intervalo_valido(alturas, i, j) : j - i }
 *   !! FALTA AÑADIR AL INVARIANTE valido Y CUANTOS SEGUIDOS !!
 *
 *   FUNCIÓN DE COTA
 *   -----
 *   C = N - fin
 *   - C >= 0 al empezar el bucle (C = N - fin = N - 1 >= 0 porque N > 0 al empezar el bucle)
 *   - C decrece al avanzar el bucle (se incrementa fin --> disminuye N - fin)
 *
 *   COSTE
 *   -----
 *   El coste asintótico en tiempo de la función es del orden O(N), dado que hay que recorrer 1 vez todo el vector y se realizan
 *   operaciones de coste O(1).
 *
 *   NOTA: El bucle interior no supone un coste asintótico añadido ya que es el que incrementa el contador y por lo tanto hace que
 *         el bucle exterior de menos vueltas.
 */

```

```

const int MAX_SEGUIDOS = 3;

// función que resuelve el problema
int resolver(vector<int> const& alturas) {
    if (alturas.size() == 0)
        return 0;

    int ini, fin = 1;
    int cuantos_seguidos;
    bool valido;
    int max_long = 1;

    while (fin < alturas.size()) {

        // reiniciamos intervalo:
        ini = fin - 1;
        cuantos_seguidos = 1;
        valido = true;

        while (valido && fin < alturas.size())
            if (alturas[fin - 1] < alturas[fin])
                if (cuantos_seguidos == MAX_SEGUIDOS - 1)
                    valido = false;
                else {
                    ++cuantos_seguidos;
                    ++fin;
                }
            else {
                ++fin;
                cuantos_seguidos = 1;
            }

        max_long = max(max_long, fin - ini); // actualizamos solución

    }

    return max_long;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    vector<int> alturas(n);
    for (int i = 0; i < n; ++i)
        cin >> alturas[i];

    int sol = resolver(alturas);

    // escribir sol
    cout << sol << '\n';
}

```

Un móvil nuevo

Enunciado:

Queremos cambiar de móvil y hemos buscado los precios de los teléfonos que ofrecen un par de marcas. Tenemos una lista con los precios que figuran en el catálogo de la marca *A* y otra lista con los precios de los teléfonos de la marca *B*. Ambas listas están ordenadas por precio de menor a mayor y en la misma lista no hay precios repetidos.

Para hacernos una idea de la variedad de precios que hay, queremos obtener una lista con todos los precios que hay en una u otra lista ordenados de menor a mayor precio. Si un precio figura en las dos listas lo pondremos una única vez en nuestra lista final.

Requisitos de implementación.

El problema debe resolverse con una función que recibe dos vectores con los precios que ofrecen las dos marcas de teléfonos. El resultado de la función es otro vector con los precios de ambas listas.

La función debe tener un coste lineal en el número de elementos de las dos listas. Para ello solo se puede acceder a cada dato del vector una vez.



Entrada

La entrada comienza con un valor entero que indica el número de casos de prueba. Cada caso de prueba consta de tres líneas. La primera línea tiene dos valores que indican el número de precios de cada una de las listas. Las dos líneas siguientes tienen los precios de cada una de las listas.

El número de precios de cada lista es mayor o igual que cero y menor que 100.000. Los precios son enteros positivos.

Salida

Para cada caso de prueba se escribe en una línea la lista de precios resultante ordenada de menor a mayor y sin repetidos.

Entrada de ejemplo

```
3
4 5
3 6 10 20
1 2 10 15 20
0 2

5 10
3 3
1 2 3
4 5 6
```

Salida de ejemplo

```
1 2 3 6 10 15 20
5 10
1 2 3 4 5 6
```

Autor: Isabel Pita

Solución:

```
✉// Beatriz Espinar Aragón
└// F19
```

```

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *  ACLARACIONES:
 *  - n1 = tamaño vct1
 *  - n2 = tamaño vct2
 *  - n = tamaño sol
 *
 *  ESPECIFICACIÓN FORMAL
 *  -----
 *  P = { 0 <= n1, n2 < 100.000 && elementosPositivos(vct1, n1) && elementosPositivos(vct2, n2) && ordenado(vct1, n1) && ordenado(vct2, n2) }
 *  fun todosLosPrecios(vct1[0..n1] de ent, vct2[0..n2] de ent) dev sol[0..n] de ent
 *  Q = { ordenado(sol, n) && ParaTodo w : 0 <= w < n : pertenece(sol, vct1, n1, w) || pertenece(sol, vct2, n2, w) }
 *
 *  donde se utilizan los siguientes predicados auxiliares:
 *  - elementosPositivos(v, n) = (ParaTodo w : 0 <= w < n : v[w] > 0)
 *  - ordenado(v, n) = (ParaTodo i : 0 <= i < n - 1 : v[i] < v[i + 1])
 *  - pertenece(v, v', n', w) = (Existe i : 0 <= i < n' : v'[i] = v[w])
 */

```

```

/* El coste asintótico de esta función será del orden O(N), donde N = max{n1, n2}, siendo n1 y n2 los
 * tamaños de los vectores. Esto es porque se han de recorrer todas las posiciones de ambos vectores
 * (se recorren a la vez), por lo que el que determinará el coste será aquel que tenga más posiciones,
 * es decir, mayor tamaño */
vector<int> todosLosPrecios(vector<int> const& vct1, vector<int> const& vct2) {

    int cont1 = 0, cont2 = 0;
    vector<int> sol;

    while (cont1 < vct1.size() && cont2 < vct2.size()) { // da min{n1, n2} vueltas
        if (vct1[cont1] < vct2[cont2]) {
            sol.push_back(vct1[cont1]);
            ++cont1;
        }
        else if (vct1[cont1] == vct2[cont2]) {
            sol.push_back(vct1[cont1]);
            ++cont1;
            ++cont2;
        }
        else {
            sol.push_back(vct2[cont2]);
            ++cont2;
        }
    }

    // da 0 vueltas si min{n1, n2} = n1
    // da n1 - n2 vueltas eoc
    while (cont1 < vct1.size()) { // si quedan elementos en el primer vector los añadimos en orden
        sol.push_back(vct1[cont1]);
        ++cont1;
    }

    // da 0 vueltas si min{n1, n2} = n2
    // da n2 - n1 vueltas eoc
    while (cont2 < vct2.size()) { // si quedan elementos en el segundo vector los añadimos en orden
        sol.push_back(vct2[cont2]);
        ++cont2;
    }

    return sol;
}

```

```

// función que resuelve el problema
vector<int> resolver(vector<int> const& vct1, vector<int> const& vct2) {
    return todosLosPrecios(vct1, vct2);
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    int n1; cin >> n1;
    int n2; cin >> n2;
    vector<int> vct1(n1), vct2(n2);
    for (int i = 0; i < n1; ++i)
        cin >> vct1[i];
    for (int i = 0; i < n2; ++i)
        cin >> vct2[i];

    vector<int> sol = resolver(vct1, vct2);

    // escribir sol
    for (int i = 0; i < sol.size(); ++i)
        cout << sol[i] << " ";
    cout << '\n';
}

```

Rescate aéreo 1

Enunciado:

Después de sufrir un devastador ataque alienígena, el alto mando ha recibido una llamada de rescate de la ciudad de Nueva York. El grupo de supervivientes se encuentra detrás de la línea de rascacielos de la ciudad. Al otro lado de los edificios se encuentra vigilando una nave extraterrestre. Para evitar que el transporte aéreo encargado del rescate sea detectado por la nave enemiga, hemos encontrado la secuencia más larga de edificios cuya altura es estrictamente mayor que la de nuestro transporte. El alto mando ya ha avisado a los supervivientes para que se dirijan a este punto de encuentro. Suponemos que todos los edificios tienen la misma anchura.

Requisitos de implementación.

Indicar el coste de la solución obtenida.

La función que resuelve el problema debe recibir los datos en un vector y devolver el comienzo y final del intervalo.

Entrada

La entrada comienza con el número de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica el número de edificios de la línea n , seguido de la altura del transporte t . En la segunda se indica la altura de cada edificio de la línea.

Se supone que la altura del transporte es menor que la altura de alguno de los edificios considerados y por lo menos existe un edificio.

Salida

Para cada caso de prueba se escriben en una línea el comienzo y el final del intervalo. En caso de existir dos intervalos iguales se eligirá el de la izquierda.

Entrada de ejemplo

2
10 5
3 6 4 8 9 8 7 2 8 9
7 5
8 8 8 2 3 9 9

Salida de ejemplo

```
3 6  
0 2
```

Autor: Isabel Pita.

Solución:

```
// Beatriz Espinar Aragón  
// F19  
  
#include <iostream>  
#include <iomanip>  
#include <fstream>  
#include <vector>  
using namespace std;  
  
/*  
 * ACLARACIONES:  
 * - n = tamaño del vector alturas  
 *  
 * ESPECIFICACIÓN FORMAL  
 * -----  
 * P = { n >= 1 & (Existe w : 0 <= w < n : alturas[w] > t) }  
 * fun mayorIntervalo(alturas[0..n) de ent, t : ent) dev principio : ent, final : ent  
 * Q = { intervalo(principio, final, alturas, n, t) & maximoIzq(principio, final, alturas, n, t) }  
 *  
 * donde se utilizan los siguientes predicados auxiliares:  
 * - intervalo(ini, fin, vct, n, alt) = (0 <= ini <= fin < n & ParaTodo i : ini <= i <= fin : vct[i] > alt)  
 * - maximoIzq(ini, fin, vct, n, alt) = (ParaTodo i,j : 0 <= i <= j < n & intervalo(i, j, vct, n, alt) : (fin - ini > j - i) || (fin - ini = j - i --> ini < i))  
 */  
  
/* El coste asintótico de esta función está en el orden O(n), donde n es el número de alturas.  
 * Esto es porque recorremos 1 vez el vector, pasando 1 vez por cada uno de sus elementos. Además,  
 * vemos que la instrucción crítica se ejecutará n veces en total, dado que el bucle finaliza cuando  
 * cont >= n, y el valor inicial es cont = 0 */  
pair<int, int> mayorIntervalo(vector<int> const& alturas, int t) {  
  
    int mejorPrincipio = 1, mejorFinal = 0; // para que al inicio el intervalo mejor tenga un tamaño "-1"  
    int auxPrincipio, auxFinal;  
    int cont = 0;  
  
    while (cont < alturas.size()) {  
        if (alturas[cont] > t) {  
            auxPrincipio = cont;  
            auxFinal = cont;  
            ++cont;  
            // El bucle siguiente no modifica el coste de la función, puesto que avanza el contador y por  
            // tanto cuando salgamos de él, quedarán menos vueltas que dar en el bucle "grande"  
            while (cont < alturas.size() && alturas[cont] > t) {  
                ++auxFinal;  
                ++cont; // podríamos tomar esta instrucción como instrucción crítica  
            }  
            if (auxFinal - auxPrincipio > mejorFinal - mejorPrincipio) {  
                mejorPrincipio = auxPrincipio;  
                mejorFinal = auxFinal;  
            }  
        }  
        else  
            ++cont;  
    }  
  
    return { mejorPrincipio, mejorFinal };  
}
```

```
// función que resuelve el problema
pair<int, int> resolver(vector<int> const& alturas, int t) {
    return mayorIntervalo(alturas, t);
}
```

```
// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    int t; cin >> t;
    vector<int> alturas(n);
    for (int i = 0; i < n; ++i)
        cin >> alturas[i];

    pair<int, int> sol = resolver(alturas, t);

    // escribir sol
    cout << sol.first << " " << sol.second << '\n';
}
```

¿Cómo van las encuestas?

Enunciado:

En estadística un valor *atípico* es una observación que es numéricamente distante del resto de los datos. Las estadísticas derivadas de los conjuntos de datos que incluyen valores atípicos serán normalmente engañosas (wikipedia).

El jefe quiere saber la media, representada por la suma de todos los valores y el número de valores, de los datos obtenidos en un encuesta. Nos ha encargado que no consideremos el valor mínimo ya que este corresponde a las personas a las que no les gusta la empresa y por lo tanto su opinión no le interesa.



Requisitos de implementación.

La función que resuelve el problema debe recibir los datos en un vector y devolver la suma de todos los datos no *atípicos* y el número de datos considerados en la suma.

La función que resuelve el problema debe tener un único bucle. Es decir, el vector se recorre solo una vez.

Entrada

La entrada comienza con el número de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera línea se indica el tamaño del vector, en la segunda los valores separados por blancos.

El número de valores del vector es mayor que cero y menor que 1000. Los valores del vector se pueden guardar en una variable de tipo `int`.

Salida

Para cada caso de prueba se escribe en una línea el valor de la suma pedida seguida del número de valores sumados.

Entrada de ejemplo

```
5
6
5 -3 5 4 -3 -3
4
9 9 9 1
8
5 5 1 3 3 10 10 4
3
1 1 1
3
2147483646 2147483647 2147483647
```

Salida de ejemplo

```
14 3
27 3
40 7
0 0
4294967294 2
```

Autor: Isabel Pita

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *  ACLARACIONES
 *  N = vct.size()
 *
 *  ESPECIFICACIÓN
 *  -----
 *  P = { 0 < N < 1.000 }
 *  fun resolver(vct[0..N] de ent) dev suma : ent, numSumandos : ent
 *  Q = { suma = SUM i : 0 <= i < N & !minimo(vct, N, i) : vct[i] & numSumandos = #i : 0 <= i < N : !minimo(vct, N, i) }
 *
 *  donde minimo(v, n, i) es un predicado auxiliar:
 *  minimo(v, n, i) = (ParaTodo w : 0 <= w : n : v[w] >= v[i])
 *
 */

/* El coste asintótico de la función es del orden O(N), donde N es el número de valores. Esto es porque
 * la instrucción crítica se llega a ejecutar en el caso peor, N veces (tantas como veces se ejecuta el bucle
 * for). Nota: todas las instrucciones interiores del bucle son de coste constante O(1) */
// función que resuelve el problema
pair<long long int, int> resolver(vector<int> const& vct) {

    int minActual = vct[0];
    long long int suma = 0;
    int numSumandos = 0;
    int numMinimos = 1;
```

```

for (int i = 1; i < vct.size(); ++i) {
    if (vct[i] < minActual) { // podemos elegir esta como la instrucción crítica, por ejemplo
        suma += numMinimos * minActual;
        numSumandos += numMinimos;
        minActual = vct[i];
        numMinimos = 1;
    }
    else if (vct[i] > minActual) {
        suma += vct[i];
        ++numSumandos;
    }
    else
        ++numMinimos;
}

return { suma, numSumandos };
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    int N; cin >> N;
    vector<int> vct(N);
    for (int i = 0; i < N; ++i)
        cin >> vct[i];

    pair<long long int, int> sol = resolver(vct);

    // escribir sol
    cout << sol.first << " " << sol.second << '\n';
}

```

Reservas de elefantes

Enunciado:

El parque nacional de Chobe, en la región noroeste de Botswana, es famoso por su alta densidad de elefantes. Los encargados del parque llevan muchos años recopilando datos sobre el número de nacimientos de elefantes en la región. Sin embargo, los visitantes del parque no suelen estar interesados en los nacimientos de un determinado año, sino que preguntan por el número de nacimientos entre dos fechas, lo que obliga a los encargados a sumar los datos entre las dos fechas pedidas. Cansados ya de tanto sumar, han decidido que deben mantener las sumas ya realizadas y de esta forma solo tendrán que consultar el dato. Pero mantener todos los intervalos de fechas no resulta viable, por lo que después de pensarla un poco han decidido que con tener los datos acumulados de los nacimientos de todos los años anteriores es suficiente.



Requisitos de implementación.

Los datos deben prepararse para que resolver cada pregunta tenga un coste constante. La preparación de los datos tiene coste lineal en el número de datos que tiene el parque.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso se escribe en varias líneas. En la primera se muestra el primer y último año ($0 < p \leq u \leq 100.000$) en los que se han tomado datos de los nacimientos de elefantes. En la segunda línea se muestran los $n = u-p+1$ nacimientos ocurridos durante estos años ($0 \leq k < 2^{30}$). En la línea siguiente se indican el número de preguntas que realizan los visitantes ($0 < m \leq 100.000$) y en las m líneas siguientes se muestran dos años ($0 \leq a \leq b < 1.000$) que representan el intervalo en el que se debe calcular el número de nacimientos de elefantes.

El final de los casos se marca con dos ceros que no deben tratarse.

Salida

Para cada caso de entrada se escriben tantas líneas como preguntas haya en el caso. En cada línea se escribe el número de nacimientos ocurridos durante los años que se indican en la pregunta. Ambos años incluidos.

Cada caso termina con una línea con tres guiones (---).

Entrada de ejemplo

```
1950 1955
4 1 2 0 3 6
3
1950 1953
1952 1955
1951 1954
2010 2016
4 2 1 6 4 4 2
2
2014 2014
2010 2016
0 0
```

Salida de ejemplo

```
7
11
6
---
4
23
---
```

Autor: Isabel Pita

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;
```

```

/*
 *  ACLARACIONES:
 *  N = births.size()
 *  M = requests.size() = sol.size()
 *
 *  ESPECIFICACIÓN FORMAL
 *  -----
 *  P = { (0 < first <= last <= 100.000) & (N = first - last + 1) & (ParaTodo w : 0 <= w < N : 0 <= births[w] < 2^30) &
 *        & (0 < m <= 100.000) & ??? (ParaTodo a : 0 <= a < M : first <= a.first <= a.second <= last) ??? }
 *  fun resolver(first : ent, births[0..N] de ent, requests[0..M] de ent) dev sol[0..M] de ent
 *  Q = { ParaTodo i : 0 <= i < M : sol[i] = SUM w : (requests[i].second - first) <= w <= (requests[i].first - first) : births[w] }
 *
 *  NOTA: el parámetro last no se conoce en la función pero es relevante para la definición de la precondition
 */

```

/* El coste asintótico en tiempo de esta función será del orden $O(N + M) = O(\max\{N, M\})$, donde N es el número de años en los que se han tomado datos de los nacimientos de elefantes, y M es el número de preguntas que se deben resolver. Este coste lineal es debido a que preparar los datos para las preguntas requiere de un for que recorra 1 vez el vector `births` haciendo operaciones constantes $O(1)$ sobre cada elemento, y posteriormente se deben procesar las M preguntas, siendo el coste de resolver cada pregunta también constante. */

```

// función que resuelve el problema
vector<long long int> resolver(int first, vector<int> const& births, vector<pair<int, int>> const& requests) {

    // creamos vector de nacimientos de los años anteriores:
    vector<long long int> acumulados(births.size() + 1);
    acumulados[0] = 0;
    for (int i = 1; i < acumulados.size(); ++i) // N vueltas
        acumulados[i] = acumulados[i - 1] + births[i - 1];

    // resolvemos las requests:
    vector<long long int> sol(requests.size());
    for (int i = 0; i < requests.size(); ++i) // M vueltas
        sol[i] = acumulados[requests[i].second - first + 1] - acumulados[requests[i].first - first]; // coste O(1)

    return sol;
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int primero; cin >> primero;
    int ultimo; cin >> ultimo;

    if (primero == 0 && ultimo == 0)
        return false;

    int numNacimientos = ultimo - primero + 1;
    vector<int> nacimientos(numNacimientos);
    for (int i = 0; i < numNacimientos; ++i)
        cin >> nacimientos[i];
    int numPreguntas; cin >> numPreguntas;
    vector<pair<int, int>> preguntas(numPreguntas);
    for (int i = 0; i < numPreguntas; ++i) {
        cin >> preguntas[i].first;
        cin >> preguntas[i].second;
    }

    vector<long long int> sol = resolver(primer, nacimientos, preguntas);

    // escribir sol
    for (int i = 0; i < numPreguntas; ++i)
        cout << sol[i] << '\n';
    cout << "---" << '\n';

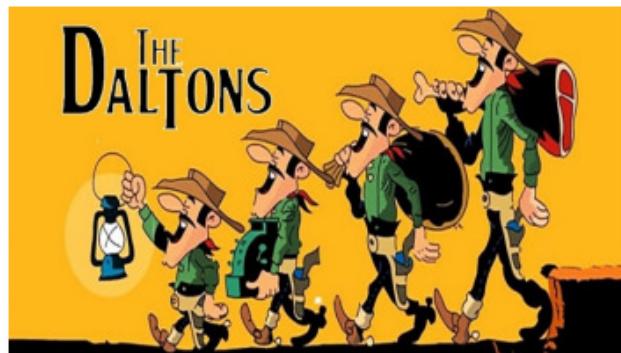
    return true;
}

```

Los hermanos Dalton

Enunciado:

Los Dalton (o también *hermanos Dalton*) son los acérrimos enemigos de Lucky Luke, el personaje creado por Maurice de Bévere (Morris).



Es ampliamente conocido su aspecto y la forma de colocarse en las viñetas. Por un lado, todos ellos tienen la misma apariencia, siendo la altura su única diferencia física. Por otro lado, en las viñetas siempre se colocan por orden de altura, formando una graciosa estampa reconocible al instante. Aunque en los cómics finalmente hay cuatro hermanos Dalton, existe la creencia de que Morris hizo bocetos en los que aparecían muchos más hermanos, todos exactamente iguales salvo por su altura. Hoy coleccionistas de todo el mundo buscan incansablemente esos bocetos. Para facilitar la identificación se está trabajando en un software que reconozca si una viñeta puede o no ser una estampa de los Dalton. Suponiendo que mediante una extracción de siluetas se hayan conseguido las alturas de todos los personajes que aparecen en un dibujo dado, habría que decidir si pueden o no ser ellos.

Separa en una función la funcionalidad clave del ejercicio e indica en comentarios su *especificación (precondición y postcondición)*. Indica también el *invariante y la función de cota* del bucle.

Entrada

La entrada estará compuesta por la descripción de varias viñetas. Cada una de ellas aparece en dos líneas. La primera tiene el número N de personas que hay en la viñeta (como mínimo dos, como mucho 100.000). La segunda tiene las N alturas de cada uno, empezando por el personaje de la izquierda y terminando por el de más a la derecha. El sistema de extracción de siluetas nos da las alturas en números enteros entre 1 y 10^6 . La entrada termina con una viñeta sin personajes, que no deberá procesarse.

Salida

Se escribirá una línea por cada viñeta, indicando si todos los personajes que hay en ella pueden ser los Dalton (DALTON) o no (DESCONOCIDOS).

Entrada de ejemplo

```
4
1 2 3 4
5
10 1 2 3 4
4
1 1 2 2
0
```

Salida de ejemplo

```
DALTON
DESCONOCIDOS
DESCONOCIDOS
```

Solución:

```
// Beatriz Espinar Aragón
// F19
```

```

/*
 *  ACLARACIONES:
 *  N = vct.size()
 *
 *  ESPECIFICACIÓN
 *  -----
 *  P = { 2 <= N <= 100.000 & ParaTodo w : 0 <= w < N : 0 < vct[w] <= 1.000.000 }
 *  fun resolver(vct[0..N) de ent) dev dalton : bool
 *  Q = { dalton = (ParaTodo i : 0 <= i < N - 1 : vct[i] < vct[i + 1]) || (ParaTodo j : 0 <= j < N - 1 : vct[j] > vct[j + 1]) }
 */

/* El coste asintótico de esta función es del orden O(N), donde N es el número de personas
 * de la viñeta. La instrucción crítica se ejecuta, en el caso peor (= si son Dalton), N veces.
 * Nota: en el resto de casos el bucle dará menos vueltas y el coste será menor */
// función que resuelve el problema
bool resolver(vector<int> const& vct) {

    bool dalton = true;
    int cont = 0;

    if (vct[cont] > vct[cont + 1])
        while (dalton && cont < vct.size() - 1) {
            if (vct[cont] <= vct[cont + 1])
                dalton = false;
            ++cont; // instrucción crítica (por ejemplo)
        }
    else if (vct[cont] < vct[cont + 1])
        while (dalton && cont < vct.size() - 1) {
            if (vct[cont] >= vct[cont + 1])
                dalton = false;
            ++cont;
        }
    else // caso mejor (los dos primeros son iguales)
        dalton = false;

    return dalton;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int N; cin >> N;
    if (N == 0)
        return false;
    vector<int> vct(N);
    for (int i = 0; i < N; ++i)
        cin >> vct[i];

    bool sol = resolver(vct);

    // escribir sol
    if (sol) cout << "DALTON";
    else cout << "DESCONOCIDOS";
    cout << '\n';

    return true;
}

```

Aavitallamiento en las etapas ciclistas

Enunciado:

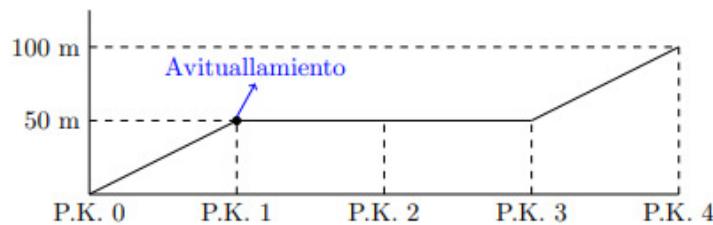
En toda etapa ciclista que se precie hay un momento en el que se permite a los participantes recuperar fuerzas recogiendo comida que voluntarios apostados en los márgenes de la carretera les dan dentro de unas bolsas.

La organización de esas etapas debe decidir en qué punto kilométrico colocan la zona de avituallamiento (el lugar en el que se les proporciona las bolsas). El sentido común indica que:

- Debe ser en una zona llana (si es subida los ciclistas estarán más concentrados en pedalear que en comer; si es bajada estarán atentos a no caerse con el incremento de velocidad).
- La zona llana debe ser la más larga que haya en toda la etapa, para darles tiempo a comer con calma.

Para tomar esta decisión, te han pedido ayuda. Ellos te darán la altura sobre el nivel del mar a la que empieza cada uno de los kilómetros de la etapa, y tendrás que decidir en qué punto kilométrico se deberán colocar los encargados del avituallamiento.

Como ejemplo, supongamos una etapa de 4 kilómetros, en la que la salida, situada en el punto kilométrico (P.K.) 0, esté sobre el nivel del mar, los P.P.K.K. 1, 2 y 3 estén a una altura de 50 metros y, por último, la meta (P.K. 4), esté a 100 metros de altitud. Con esta configuración de etapa, existen dos kilómetros llanos (que comienzan en los P.P.K.K. 1 y 2), por lo que el avituallamiento se colocará en el P.K. 1, para que los corredores tengan dos kilómetros completos para comer.



Entrada

La entrada está compuesta de múltiples etapas, cada una en una línea. Una etapa está formada por una secuencia de al menos dos números mayores o iguales que cero separados por espacios. Cada uno de esos números representa la altura sobre el nivel del mar al principio del kilómetro, siendo el primer número la altura al principio del primer kilómetro (P.K. 0), el segundo al principio del segundo kilómetro, etc. Cada etapa termina con un -1 que no debe tenerse en cuenta en el cálculo. La última altura válida leída es la altura a la que se encuentra la meta. Todas las etapas tienen menos de 100.000 kilómetros.

La entrada termina con una "etapa vacía", es decir una línea que contiene únicamente un -1.

Salida

Para cada etapa se escribirá una línea donde aparecerá el número de kilómetros llanos que se dedicarán al avituallamiento. Si en la etapa no hay ninguna zona llana, se escribirá HOY NO COMEN.

Entrada de ejemplo

```
0 50 50 50 100 -1
10 10 -1
0 5 -1
0 50 50 100 100 -1
0 50 50 3 3 3 -1
-1
```

Salida de ejemplo

```
2
1
HOY NO COMEN
1
2
```

Notas

Para este problema debes implementar una función del tipo:

```
int tamSegmentoMaximo(int v[], int n);
```

que devuelva la longitud del segmento más grande con todos los elementos iguales. La salida será HOY NO COMEN si su salida es 0, y el resultado de restarle uno al valor devuelto en caso contrario.

No olvides escribir la precondición, postcondición e invariante en comentarios. Para que te sea más fácil, es posible que quieras definir `iguales(v, a, b)` que es cierto si todos los elementos del vector `v` entre `a` y `b` son iguales.

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ACLARACIONES:
 *   N = vct.size()
 *
 *   ESPECIFICACIÓN FORMAL
 *
 *   -----
 *   P = { 2 <= N < 100.000 & (ParaTodo w : 0 <= w < N : alturas[w] >= 0) }
 *   fun resolver(alturas[0..N) de ent) dev kms : ent
 *   Q = { kms = maxIntervalo(alturas, N) }
 *
 *   donde se utilizan los siguientes predicados auxiliares:
 *   - maxIntervalo(v, n) = (max ini,fin : 0 <= ini < fin & llano(v, ini, fin) : fin - ini)
 *   - llano(v, ini, fin) = (ParaTodo i : ini <= i < fin : v[i] = v[i + 1])
 *
 */

/* El coste asintótico en tiempo de esta función es del orden O(N), donde N es el número de kilómetros de la etapa. Este coste lineal se debe a que recorremos el vector de alturas pasando por cada elemento 1 única vez y haciendo operaciones de coste constante O(1) sobre él.
 * NOTA: El bucle interior del bucle grande no supone un aumento del coste puesto que incrementa el valor del contador, disminuyendo así el número de veces que se ejecuta el bucle exterior. */

// función que resuelve el problema
int resolver(vector<int> const& alturas) {

    int mejorIni = 0, mejorFin = 0;
    int iniAux, finAux;
    int cont = 0;

    while (cont < alturas.size() - 1) {

        if (alturas[cont] == alturas[cont + 1]) {
            iniAux = cont;
            finAux = cont + 1;
            ++cont;
            // Mientras sea zona llana:
            while (cont < alturas.size() - 1 && alturas[cont] == alturas[cont + 1]) {
                ++finAux;
                ++cont;
            }
            // Si el intervalo es mayor que el mejor que teníamos hasta el momento, actualizamos
            if (finAux - iniAux > mejorFin - mejorIni) {
                mejorIni = iniAux;
                mejorFin = finAux;
            }
        }
    }
}
```

```

        ++cont;
    }

    return { mejorFin - mejorIni };
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int aux; cin >> aux;
    if (aux == -1)
        return false;
    vector<int> alturas;
    while (aux != -1) {
        alturas.push_back(aux);
        cin >> aux;
    }

    int sol = resolver(alturas);

    // escribir sol
    if (sol == 0) cout << "HOY NO COMEN";
    else cout << sol;
    cout << '\n';

    return true;
}

```

Viajes a Marte

Enunciado:

Los vuelos a Marte están cada vez más solicitados. Para poder cubrir la demanda, la compañía ha encargado unos vehículos espaciales con más filas de asientos. El constructor ha propuesto disminuir la distancia entre las filas de la parte posterior. De esta forma no será necesario hacer modificaciones en el diseño del aparato. El problema está en que en estas filas sólo podrán sentarse personas *bajitas*. La compañía, sin embargo, no quiere aumentar el presupuesto, por lo que ha decidido pedir a los viajeros su altura cuando compran el billete y a la hora de embarcar llama primero a los más *bajos* para que ocupen las filas posteriores.



Tu misión es dividir a los viajeros entre *bajos* y *altos* en función de la altura que te digan. Si la altura de algún viajero coincide con la altura establecida se le considerará *bajo*. Debes también indicar el número de viajeros *bajos* del vuelo.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de $n+1$ líneas. En la primera se indica el número n de viajeros y la altura establecida para diferenciarlos. En cada una de las líneas siguientes aparece el nombre de un viajero y su altura.

El número de viajeros es mayor o igual que 0 y menor que 150.000. Los nombres de los viajeros son cadenas de caracteres sin blancos. La altura son números enteros positivos.

Salida

Para cada caso de prueba se escriben dos líneas. La primera comienza con **Bajos**: seguido de los nombres de los viajeros *bajos*. La segunda comienza con **Altos**: seguido de los nombres de los viajeros *altos*.

Para poder comparar tu salida con la del juez los nombres de los viajeros se mostrarán en orden alfabético, aunque esto no debe alterar el algoritmo empleado para resolver el problema, sino solo la salida de los datos.

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

/*
 * ACLARACIONES
 * -----
 * N = vectorViajeros.size() (= número de viajeros)
 *
 * ESPECIFICACIÓN
 * -----
 * P = { 0 <= N < 150.000 } \ ParaTodo w : 0 <= w < N : v[w] > 0 /\ V = V }
 * proc resolver(E/S v[0..N) de <string, ent>, alt : ent, S primerAlto : ent)
 * Q = { v ∈ Perm(V) } \ 0 <= primerAlto <= N /\ ParaTodo i : 0 <= i < primerAlto : bajo(v, i, alt) /\ ParaTodo j : primerAlto <= j < N : !bajo(v, j, alt) }

 * Donde se utilizan el siguiente predicado auxiliar:
 *   - bajo(v, i, a) = (v[i] <= a)

 * COSTE
 * -----
 * El coste asintótico en tiempo de esta función es del orden O(N), dado que el bucle se ejecutará N veces en cualquier caso.
 * Mirando los casos extremos, si todos son bajos cont no varía, pero primerAlto recorre todo el vector. Si todos son
 * altos primerAlto no varía, pero cont recorre todo el vector. Por lo tanto, el coste será lineal.
 * NOTA: la función intercambiar() no supone un coste asintótico añadido dado que sus instrucciones son de coste O(1) y no hay
 * ningún bucle
 *
 * INVARIANTE
 * -----
 * I = (v ∈ Perm(V) ) \ 0 <= primerAlto <= cont <= N /\ ParaTodo i : 0 <= i <= primerAlto : bajo(v, i, alt) /\ ParaTodo j : primerAlto <= j < N : !bajo(v, j, alt)
 * B = cont > primerAlto
 *
 * FUNCIÓN DE COTA
 * -----
 * C = cont - primerAlto
 *   - C >= 0 al principio del bucle (cont - primerAlto = N - 1 - 0 = N - 1 >= 0 (N > 0))
 *   - C decrece con cada vuelta del bucle (cont disminuye o primerAlto aumenta --> cont - primerAlto decrece)
 */
using viajeros = pair<string, int>;

// Esta función es de coste O(1)
void intercambiar(vector<viajeros>& v, int p, int q) {
    viajeros aux = v[q];
    v[q] = v[p];
    v[p] = aux;
}

// función que resuelve el problema
void resolver(vector<viajeros>& v, int alt, int& primerAlto) {

    primerAlto = 0;
    int cont = v.size() - 1;
```

```

if (v.size() > 0) { // si el vector es vacío no hacemos nada

    while (cont > primerAlto)
        if (v[cont].second <= alt) { // si es bajo lo añadimos al lado de los bajos
            intercambiar(v, cont, primerAlto);
            ++primerAlto;
        }
        else // si es alto seguimos avanzando
            --cont; // si tomamos esta instrucción crítica (si todos son altos), se ejecuta N veces

    if (v[cont].second <= alt) // si el último es bajo, el primer alto será el siguiente
        ++primerAlto;

}

}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    if (!std::cin)
        return false;
    int altura; cin >> altura;

    vector<viajeros> vectorViajeros(n);
    for (int i = 0; i < n; ++i)
        cin >> vectorViajeros[i].first >> vectorViajeros[i].second;

    // calcular sol
    int primerAlto;
    resolver(vectorViajeros, altura, primerAlto);

    // ordenar sol
    sort(vectorViajeros.begin(), vectorViajeros.begin() + primerAlto, [](viajeros v1, viajeros v2) { return v1.first < v2.first; });
    sort(vectorViajeros.begin() + primerAlto, vectorViajeros.end(), [](viajeros v1, viajeros v2) { return v1.first < v2.first; });

    // escribir sol
    cout << "Bajos: ";
    for (int i = 0; i < primerAlto; ++i)
        cout << vectorViajeros[i].first << " ";
    cout << '\n' << "Altos: ";
    for (int i = primerAlto; i < n; ++i)
        cout << vectorViajeros[i].first << " ";
    cout << '\n';

    return true;
}

```

Cintas de colores

Enunciado:

Tengo una caja llena de cintas de colores rojas, azules y verdes. Las he estado midiendo y ahora quiero ordenarlas por colores, para poder saber fácilmente las longitudes que tengo de cada color.

Dada una lista de todas las cintas con sus longitudes, sin ningún orden, debes ordenarlas de forma que al principio de la lista aparezcan las cintas azules, a continuación las verdes y por último las rojas. Dentro de cada grupo no es necesario que las cintas aparezcan en un orden determinado.



Requisitos de implementación.

Se debe implementar una función que reciba un vector con las cintas identificadas por su color y su longitud y modifique el vector para dejar al principio las cintas azules, en el medio las cintas verdes y por último las cintas rojas. La función devolverá dos índices indicando donde empiezan y terminan las cintas verdes. El coste de la función debe ser lineal en el número de cintas.

Debe emplearse el algoritmo de partición con dos índices.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de dos líneas. En la primera se indica el número n de cintas. En la siguiente se muestra el color de cada cinta seguido de su longitud.

El número de cintas es mayor o igual que 0 y menor que 300.000. Los colores de las cintas se identifican por su primer carácter: **a** para las cintas azules, **v** para las cintas verdes y **r** para las cintas rojas. Las longitudes son números enteros positivos.

Salida

Para cada caso de prueba se escriben tres líneas. La primera comienza con **Azules:** seguido de las longitudes de las cintas azules. La segunda comienza con **Verdes:** seguido de las longitudes de las cintas verdes. La tercera comienza con **Rojas:** seguido de las longitudes de las cintas rojas.

Para poder comparar tu salida con la del juez las longitudes se mostrarán en orden de menor a mayor, aunque esto no debe alterar el algoritmo empleado para resolver el problema, sino solo la salida de los datos.

Entrada de ejemplo

```
6
v 5 r 1 a 3 a 7 v 3 r 6
3
r 4 r 7 r 1
5
a 6 a 2 v 1 v 3 a 5
4
r 5 r 7 a 9 a 3
```

Salida de ejemplo

```
Azules: 3 7
Verdes: 3 5
Rojas: 1 6
Azules:
Verdes:
Rojas: 1 4 7
Azules: 2 5 6
Verdes: 1 3
Rojas:
Azules: 3 9
Verdes:
Rojas: 5 7
```

Autor: Isabel Pita

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

/*
 *   ACLARACIONES:
 *   - N = cintas.size()  (número de cintas)
 *
 *   ESPECIFICACIÓN FORMAL
 *   -----
 *   P = { 0 <= N < 300.000 } \ (ParaTodo w : 0 <= w < N : cintas[w].color = 'a' \vee cintas[w].color = 'v' \vee cintas[w].color = 'r') /\ 
 *       /\ (ParaTodo u : 0 <= u < N : cintas[u].longitud > 0) /\ v = V }
 *   proc resolver(E/S v@0..N) de <char, ent>, S primerVerde : ent, S ultVerde : ent)
 *   Q = { v ∈ Perm(V) /\ 0 <= primerVerde <= ultVerde <= N /\ (ParaTodo i : 0 <= i < primerVerde : cintas[i].color = 'a') /\ 
 *         /\ (ParaTodo j : primerVerde <= j < ultVerde : cintas[j].color = 'v') /\ (ParaTodo k : ultVerde <= k < N : cintas[k].color = 'r') }
 *
 *   INVARIANTE
 *   -----
 *   I = { ultVerde <= primerRojo /\ v ∈ Perm(V) /\ (ParaTodo i : 0 <= i < primerVerde : cintas[i].color = 'a') /\ (ParaTodo j : primerVerde <= j < ultVerde : cintas[j].color = 'v') /\ 
 *         /\ (ParaTodo k : primerRojo <= k < N : cintas[k].color = 'r') }
 *   B = ultVerde != primerRojo
 *
 *   FUNCIÓN DE COTA
 *   -----
 *   C = primerRojo - ultVerde
 *   - C >= 0 al principio del bucle (primerRojo - ultVerde = N - 0 = N >= 0 por la precondición)
 *   - C decrece con cada vuelta del bucle (primerRojo disminuye o ultVerde aumenta --> primerRojo - ultVerde decrece)
 *
 *   COSTE
 *   -----
 *   El coste asintótico en tiempo de esta función es del orden O(N), dado que el bucle se ejecutará N veces en cualquier caso.
 *   Mirando los casos extremos, si todas las cintas fueran verdes/azules primerRojo no varía, pero ultVerde recorre todo el vector.
 *   En caso contrario, si todas las cintas fueran rojas ultVerde no varía, pero primerRojo recorre todo el vector. Por lo tanto,
 *   el coste será lineal.
 *
 *   NOTA: la función intercambiar() no supone un coste asintótico añadido dado que sus instrucciones son de coste O(1) y no hay
 *         ningún bucle
 */
struct cinta {
    char color;
    int longitud;

    bool operator<(cinta c) {
        return longitud < c.longitud;
    }
};

// Esta función auxiliar es de coste O(1)
void intercambiar(vector<cinta>& c, int p, int q) {
    cinta aux = c[q];
    c[q] = c[p];
    c[p] = aux;
}

// función que resuelve el problema
void resolver(vector<cinta>& cintas, int& primerVerde, int& ultVerde) {
    // Inicializamos:
    primerVerde = 0;
    ultVerde = 0;
    int primerRojo = cintas.size();

    // Bucle:
    while (ultVerde != primerRojo) {

        switch (cintas[ultVerde].color) {
            case 'a': { // lo añadimos por la dch a los azules
                intercambiar(cintas, primerVerde, ultVerde);
                ++primerVerde;
                ++ultVerde;
            }
        }
    }
}
```

```

        break;
    case 'r': { // lo añadimos por la izq a los rojos
        intercambiar(cintas, ultVerde, primerRojo - 1);
        --primerRojo;
    }
        break;
    case 'v': { // lo dejamos donde está
        ++ultVerde;
    }
        break;
    }
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    if (!std::cin)
        return false;
    vector<cinta> cintas(n);
    for (int i = 0; i < n; ++i)
        cin >> cintas[i].color >> cintas[i].longitud;

    int primerVerde, ultVerde;
    resolver(cintas, primerVerde, ultVerde);

    // ordenar sol
    sort(cintas.begin(), cintas.begin() + primerVerde);
    sort(cintas.begin() + primerVerde, cintas.begin() + ultVerde);
    sort(cintas.begin() + ultVerde, cintas.end());

    // escribir sol
    cout << "Azules: ";
    for (int i = 0; i < primerVerde; ++i)
        cout << cintas[i].longitud << " ";
    cout << '\n' << "Verdes: ";
    for (int i = primerVerde; i < ultVerde; ++i)
        cout << cintas[i].longitud << " ";
    cout << '\n' << "Rojas: ";
    for (int i = ultVerde; i < n; ++i)
        cout << cintas[i].longitud << " ";
    cout << '\n';

    return true;
}

```

Releyendo un libro. Versión 1

Enunciado:

Cuando tengo un rato libre me gusta coger un libro y releer alguna de las partes que más me gustaron cuando lo leí por primera vez. Normalmente, mientras voy leyendo doy una puntuación entre 0 y 10 a cada página, de forma que cuando quiero volver a leerlo se que páginas me gustaron más. Cero significa que no tengo interés en volver a leer esa página, mientras que un 10 indica que es una de las mejores partes de la obra. Una vez que empiezo a leer en una página, continuo con las páginas siguientes sin saltarme ninguna.

Para asegurarme de que leo la mejor parte he aprendido a estimar el número de páginas que me dará tiempo a leer en el rato que tengo. Siempre busco que las páginas sean lo más interesantes posible. El interés de varias páginas se calcula como la suma del interés de cada página. Si hay varios intervalos igual de interesantes elijo el que esté más avanzado en el libro. Recuerda que las páginas que leo son siempre consecutivas.



Requisitos de implementación.

La función que resuelve el problema debe tener un coste lineal respecto al número de páginas del libro. Puede implementarse más de un bucle, siempre que el coste sea lineal respecto al número de páginas del libro.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica el número de páginas del libro ($1 < n < 200.000$), seguido de la cantidad de páginas que me dará tiempo a leer ($1 < L < n$). En la segunda línea se indica la puntuación que he dado a cada página del libro ($0 \leq p \leq 10$). La entrada finaliza con dos ceros.

Salida

Para cada caso de prueba se escribe la página en que debo empezar a leer el libro para que al leer L páginas empezando en ella, lea siempre las páginas más interesantes. Si existen dos bloques de páginas cuya suma de puntuaciones es igual, elegiré la que se encuentre más avanzada en el libro.

Entrada de ejemplo

```
6 3
0 5 3 4 1 2
7 2
6 4 5 6 0 0 8
5 4
7 7 7 7 7
6 2
5 6 7 6 7 5
0 0
```

Salida de ejemplo

```
1
2
1
3
```

Autor: Isabel Pita

Solución:

```
// // Beatriz Espinar Aragón
// // F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 * ACLARACIONES:
 * - N = pags.size() (= número de páginas del libro)
 *
 * ESPECIFICACIÓN FORMAL
 * -----
 * P = { 1 < N < 200.000 /\ 1 < L < N /\ (ParaTodo w : 0 <= w < N : 0 <= pags[w] <= 10) }
 * fun resolver(pags[0..N] de ent, L : ent) dev sol : ent
 * Q = { ParaTodo i,j : 0 <= i < j <= N /\ j - i = L : (suma(pags, i, j) < suma(pags, sol, sol + L)) \vee (suma(pags, i, j) = suma(pags, sol, sol + L) /\ i <= sol) }
 *
 * donde se utiliza el siguiente predicado auxiliar:
 * - suma(v, i, j) = (SUM w : i <= w < j : v[w])
 *
 * INVARIANTE
 * -----
 * - Primer bucle:
 *   I = { u <= p + L /\ sum = suma(pags, p, u) }
 *   B = u < p + L
 * - Segundo bucle:
 *   I = { u <= N /\ sum = suma(pags, p, u) /\ maxSuma = suma(pags, sol, sol + L) /\
 *         /\ (ParaTodo i,j : 0 <= i < j <= u /\ j - i = L : (suma(pags, i, j) < suma(pags, sol, sol + L)) \vee (suma(pags, i, j) = suma(pags, sol, sol + L) /\ i <= sol) ) }
 *   B = u < N
 *
```

```

/*
 * FUNCIÓN DE COTA
 *
 * -----
 * - Primer bucle:
 *   C = (p + L) - u
 *   - C >= 0 al empezar el bucle ((p + L) - u = 0 + L - 1 = L - 1 >= 1 (por la precondición) > 0)
 *   - C decrece al avanzar el bucle (aumenta u --> disminuye (p + L) - u)
 * - Segundo bucle:
 *   C = N - u
 *   - C >= 0 al empezar el bucle (N - u = N - (p + L) = N - (0 + L) = N - L >= 1 (por la precondición) > 0)
 *   - C decrece al avanzar el bucle (aumenta u --> disminuye N - u)
 */

/*
 * COSTE
 *
 * -----
 * El coste asintótico en tiempo de la función será del orden O(N), dado que entre los dos bucles se recorre el vector
 * entero pasando por cada elemento 1 única vez. Ambos aumentan el "contador" u y esta variable puede tomar a lo sumo el
 * valor de N. Poniendo un caso límite, si L = N el primer bucle dará N vueltas, pero el segundo dará 0. Además, las operaciones
 * interiores de los bucles son todas de coste O(1), por lo que no suponen un cambio en el coste asintótico de la función.
 *
 */

// función que resuelve el problema
int resolver(vector<int> const& pags, int L) {

    // Inicializamos:
    int p = 0, u = 1;
    int sum = pags[0];
    int maxSum, sol = 0;

    // Primer tramo:
    while (u < p + L) {
        sum += pags[u];
        ++u;
    }
    // u = p + L
    maxSum = sum;

    // Bucle:
    while (u < pags.size()) {
        sum -= pags[p]; // restamos el primero
        sum += pags[u]; // sumamos el siguiente
        ++p;
        ++u;
        if (sum >= maxSum) { // si es mejor (o igual) que la solución hasta ahora actualizamos
            maxSum = sum;
            sol = p;
        }
    }

    return sol;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    int L; cin >> L;
    if (n == 0 && L == 0)
        return false;
    vector<int> pags(n);
    for (int i = 0; i < n; ++i)
        cin >> pags[i];

    int sol = resolver(pags, L);

    // escribir sol
    cout << sol << '\n';

    return true;
}

```

Releyendo un libro

Enunciado:

Cuando tengo un rato libre me gusta coger un libro y releer alguna de las partes que más me gustaron cuando lo leí por primera vez. Normalmente, mientras voy leyendo doy una puntuación entre 0 y 10 a cada página, de forma que cuando quiero volver a leerlo se que páginas me gustaron más. Cero significa que no tengo interés en volver a leer esa página, mientras que un 10 indica que es una de las mejores partes de la obra. Una vez que empiezo a leer en una página, continuo con las páginas siguientes sin saltarme ninguna.



Para asegurarme de que leo la mejor parte he aprendido a estimar el número de páginas que me dará tiempo a leer en el rato que tengo.

Siempre busco que la parte que voy a leer incluya la página que más me gusta del libro y que las demás páginas sean lo más interesantes posible. Si hay varios intervalos igual de interesantes elijo el que esté más avanzado en el libro. Recuerda que las páginas que leo son siempre consecutivas.

Requisitos de implementación.

La función que resuelve el problema debe tener un coste lineal respecto al número de páginas del libro. Puede implementarse más de un bucle, siempre que el coste sea lineal respecto al número de páginas del libro.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica el número de páginas del libro ($1 < n$), seguido de la cantidad de páginas que me dará tiempo a leer ($1 < L < n$). En la segunda línea se indica la puntuación que he dado a cada página del libro ($0 \leq p \leq 10$). La entrada finaliza con dos ceros.

Salida

Para cada caso de prueba se escribe la página en que debo empezar a leer el libro para que al leer L páginas empezando en ella, lea siempre al menos una página que tenga la máxima puntuación y además no haya otras L páginas que conteniendo una página con la puntuación máxima me gusten más. Si existen dos bloques de páginas que conteniendo la página que más me gusta, su suma de puntuaciones es igual, elegiré la que se encuentre más avanzada en el libro.

Entrada de ejemplo

```
6 3
0 5 3 4 1 2
7 2
6 4 5 6 0 0 8
5 4
7 7 7 7 7
6 2
5 6 7 6 7 5
0 0
```

Salida de ejemplo

```
1
5
1
3
```

Autor: Isabel Pita

Solución:

```

//// Beatriz Espinar Aragón
//// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *  ACLARACIONES:
 *    - N = pags.size() (= número de páginas del libro)
 *
 *  ESPECIFICACIÓN FORMAL
 *  -----
 *  P = { 1 < N /\ 1 < L < N /\ (ParaTodo w : 0 <= w < N : 0 <= pags[w] <= 10) }
 *  fun resolver(pags[0..N) de ent, L : ent) dev sol : ent
 *  Q = { ParaTodo i,j : 0 <= i < j <= N /\ j - i = L /\ tieneMaximo(pags, N, i, j) : (suma(pags, i, j) < suma(pags, sol, sol + L)) \/
 *        (suma(pags, i, j) = suma(pags, sol, sol + L) /\ i <= sol) }
 *
 *  donde se utilizan los siguientes predicados auxiliares:
 *  - tieneMaximo(v, n, i, j) = (Existe w : i <= w < j : v[w] = maximo(v, 0, n))
 *  - maximo(v, i, j) = (max w : i <= w < j : v[w])
 *
 *  INVARIANTE
 *  -----
 *  - Primer bucle:
 *    I = { u <= p + L /\ sum = suma(pags, p, u) /\ (cuantosMax = #w : p <= w < u : pags[w] = maximo(pags, p, u) )
 *    B = u < p + L
 *  - Segundo bucle:
 *    I = { u <= N /\ sum = suma(pags, p, u) /\ (maxSuma = suma(pags, sol, sol + L)) \/
 *          /\ (ParaTodo i,j : 0 <= i < j <= u /\ j - i = L /\ tieneMaximo(pags, N, i, j) : (suma(pags, i, j) < suma(pags, sol, sol + L)) \/
 *          (suma(pags, i, j) = suma(pags, sol, sol + L) /\ i <= sol) ) /\ (cuantosMax = #w : p <= w < u : pags[w] = maximo(pags, p, u) )
 *    B = u < N
 *
 *  FUNCIÓN DE COTA
 *  -----
 *  - Primer bucle:
 *    C = (p + L) - u
 *    - C >= 0 al empezar el bucle ((p + L) - u = 0 + L - 1 = L - 1 >= 1 (por la precondición) > 0)
 *    - C decrece al avanzar el bucle (aumenta u --> disminuye (p + L) - u)
 *  - Segundo bucle:
 *    C = N - u
 *    - C >= 0 al empezar el bucle (N - u = N - (p + L) = N - (0 + L) = N - L >= 1 (por la precondición) > 0)
 *    - C decrece al avanzar el bucle (aumenta u --> disminuye N - u)
 *
 *  COSTE
 *  -----
 *  El coste asintótico en tiempo de la función será del orden O(N), dado que entre los dos bucles se recorre el vector
 *  entero pasando por cada elemento 1 única vez. Ambos aumentan el "contador" u y esta variable puede tomar a lo sumo el
 *  valor de N. Poniendo un caso límite, si L = N el primer bucle dará N vueltas, pero el segundo dará 0. Además, las operaciones
 *  interiores de los bucles son todas de coste O(1), por lo que no suponen un cambio en el coste asintótico de la función.
 *
 *  NOTA: la función máximo tiene también coste O(N), pero no afecta al coste asintótico total de la función, dado que O(2*N) está en
 *        el orden de O(N) (podemos omitir la constante multiplicativa)
 */
 */

// función auxiliar que calcula el valor máximo de un vector
int maximo(vector<int> const& pags) {
    int m = pags[0];
    for (int i = 1; i < pags.size(); ++i) // da N vueltas
        if (pags[i] > m)
            m = pags[i];
    return m;
}

// función que resuelve el problema
int resolver(vector<int> const& pags, int L) {
    // Inicializamos:
    int max = maximo(pags); // coste O(N)
    int sum = 0, cuantosMax = 0;
    int maxSum = -1;
    int p = 0, u = 0, sol = 0;
}

```

```

// Primer tramo:
while (u < p + L) {
    sum += pags[u];
    if (pags[u] == max)
        ++cuantosMax;
    ++u;
}
// u = p + L
if (cuantosMax > 0) // si hay algún máximo vale el tramo
    maxSum = sum;

// Bucle:
while (u < pags.size()) {

    // restamos el primer elemento:
    sum -= pags[p];
    if (pags[p] == max) // si hemos quitado un máximo
        --cuantosMax;
    ++p;

    // sumamos el siguiente elemento:
    sum += pags[u];
    if (pags[u] == max) // si hemos añadido un máximo
        ++cuantosMax;
    ++u;

    // si el intervalo es válido (tiene al menos 1 máximo) y es mejor (o igual) que lo que teníamos hasta ahora
    if (cuantosMax > 0 && sum >= maxSum) {
        maxSum = sum;
        sol = p;
    }
}

return sol;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    int L; cin >> L;
    if (n == 0 && L == 0)
        return false;
    vector<int> pags(n);
    for (int i = 0; i < n; ++i)
        cin >> pags[i];

    int sol = resolver(pags, L);

    // escribir sol
    cout << sol << '\n';

    return true;
}

```

Invertir un número

Enunciado:

En este problema vamos a invertir los dígitos de un número.

Requisitos de implementación

Los números deben tratarse como números enteros, no como cadenas de caracteres.

Entrada

La entrada consta de una serie de casos de prueba que acaban con el número 0. Cada caso de prueba es un número entero positivo que se puede almacenar en una variable de tipo int. Los números nunca empiezan con un dígito 0, pero pueden tener el dígito 0 en otras posiciones.

Salida

Para cada caso de prueba se escribe en una línea el valor inverso al de entrada.

Entrada de ejemplo

```
2345
3
12
101010
567
0
```

Salida de ejemplo

```
5432
3
21
10101
765
```

Autor: Isabel Pita.

Solución:

```
//// Beatriz Espinar Aragón
//// F19

#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

/*
 *      ESPECIFICACIÓN FORMAL
 * -----
 *      P = { n > 0 }
 *      fun invertir(n : ent) dev <num_invertido : ent, potencia : ent>
 *      Q = { num_invertido = inverso(n) /\ potencia = min w : n / 10^w = 0 : 10^w }
 *
 *      donde inverso() indica la intuición matemática que tenemos de "número invertido"
 *
 *      RECURRENCIA
 * -----
 *      T(n) = { k_0           si n < 10
 *              { T(n / 10) + k_1   eoc
 *
 *      COSTE
 * -----
 *      Si aplicamos el Teorema de la División a partir de la función de recurrencia obtenemos que el coste asintótico
 *      de la función recursiva será del orden O(log n), donde n es el valor de entrada.
 */

```

```

// función recursiva:
pair<int, int> invertir(int n) {

    // CASO BASE:
    if (n < 10)
        return { n, 10 };

    // CASO RECURSIVO:
    else {

        // Llamada recursiva:
        pair<int, int> sol = invertir(n / 10);

        int num_invertido = n % 10; // último dígito
        num_invertido *= sol.second; // lo multiplicamos por 10^num_digits de la parte invertida
        num_invertido += sol.first; // le añadimos la parte invertida

        return { num_invertido, sol.second * 10 };
    }
}

// función que resuelve el problema
int resolver(int n) {
    return invertir(n).first;
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    if (n == 0)
        return false;

    int sol = resolver(n);

    // escribir sol
    cout << sol << '\n';
    return true;
}

```

Dígitos complementarios

Enunciado:

Diremos que un dígito es complementario de otro si la suma de ambos es 9. Así, el 0 y el 9 son dígitos complementarios, así como el 1 y el 8, el 2 y el 7 etc.

En este problema nos dan un número y debemos calcular el número formado por sus dígitos complementarios, en su mismo orden y en orden inverso.

Requisitos de implementación.

Implementar una función recursiva que reciba un número entero y devuelva el número formado por los dígitos complementarios.

Implementar otra función recursiva que reciba un número entero y devuelva el inverso del número formado por los dígitos complementarios.

Ambas funciones deben tratar cada dígito del número de entrada una sola vez. Se pueden definir más parámetros o valores de salida de la función si se considera necesario. No se puede utilizar el tipo `std::string` para resolver el ejercicio.

No olvides poner tu nombre, el usuario del juez que has utilizado hoy y el comentario con el coste de la función.

Entrada

La entrada comienza con el número de casos de prueba. A continuación cada caso se escribe en una línea y consiste en un número $0 \leq N \leq 1.000.000.000$.

Salida

Para cada caso de prueba se muestra en una línea el número obtenido sustituyendo cada dígito por su complementario, seguido del inverso del número obtenido al sustituir cada número por su inverso.

Entrada de ejemplo

```
7  
45637  
555  
2  
90  
3050217  
0  
99
```

Salida de ejemplo

```
54362 26345  
444 444  
7 7  
9 90  
6949782 2879496  
9 9  
0 0
```

Solución:

```
// Beatriz Espinar Aragón  
// F19  
  
#include <iostream>  
#include <iomanip>  
#include <fstream>  
using namespace std;  
  
/*  
 * ESPECIFICACIÓN FORMAL (complementario)  
 * -----  
 * P = { 0 <= n <= 1.000.000.000 }  
 * fun complementario(n : ent) dev comp : ent  
 * Q = { comp = complementario(n) }  
 *  
 * donde complementario(n) devuelve un n' tal que (ParaTodo i : 0 <= i (max w : n / 10^w != 0 : w) : (n / 10^i) % 10 = x --> (n' / 10^i) % 10 = 9 - x)  
 *  
 * RECURRENCIA (complementario)  
 * -----  
 * T(n) = { k_0 si n < 10  
 *          { T(n / 10) + k_1 eoc  
 *  
 * COSTE (complementario)  
 * -----  
 * Si aplicamos el Teorema de la División a partir de la función de recurrencia obtenemos que el coste asintótico  
 * de la función recursiva será del orden O(log n), donde n es el valor de entrada.  
 *  
 * ESPECIFICACIÓN FORMAL (complementario_invertido)  
 * -----  
 * P = { 0 <= n <= 1.000.000.000 }  
 * fun complementario_invertido(n : ent) dev <num_comp_inv : ent, potencia : ent>  
 * Q = { num_comp_inv = inverso(complementario(n)) /\ (n != 0 --> potencia = min w : n / 10^w = 0 : 10^w) /\ (n = 0 --> potencia = 10) }  
 *  
 * donde inverso() indica la intuición matemática que tenemos de un "número invertido"  
 * y donde complementario(n) devuelve un n' tal que (ParaTodo i : 0 <= i (max w : n / 10^w != 0 : w) : (n / 10^i) % 10 = x --> (n' / 10^i) % 10 = 9 - x)  
 *  
 * RECURRENCIA (complementario_invertido)  
 * -----  
 * T(n) = { k_0 si n < 10  
 *          { T(n / 10) + k_1 eoc  
 *  
 * COSTE (complementario_invertido)  
 * -----  
 * Si aplicamos el Teorema de la División a partir de la función de recurrencia obtenemos que el coste asintótico  
 * de la función recursiva será del orden O(log n), donde n es el valor de entrada.  
 */
```

```

// función recursiva que calcula el número complementario invertido del parámetro de entrada
pair<long long int, int> complementario_invertido(long long int n) {
    // CASO BASE:
    if (n < 10)
        return { 9 - n, 10 };

    // CASO RECURSIVO:
    else {

        // Llamada recursiva:
        pair<long long int, int> sol = complementario_invertido(n / 10);

        int num_invertido = 9 - (n % 10); // último dígito invertido
        num_invertido *= sol.second; // lo multiplicamos por 10^num_digitos de la parte invertida
        num_invertido += sol.first; // le añadimos la parte invertida

        return { num_invertido, sol.second * 10 };
    }
}

// función recursiva que calcula el complementario del número pasado como entrada
long long int complementario(long long int n) {
    // CASO BASE:
    if (n < 10)
        return 9 - n;

    // CASO RECURSIVO:
    else
        return (complementario(n / 10) * 10 + (9 - (n % 10)));
}

// función que resuelve el problema
pair<long long int, long long int> resolver(long long int n) {
    long long int comp = complementario(n);
    pair<long long int, int> inv = complementario_invertido(n);
    return { comp, inv.first };
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    long long int n; cin >> n;

    pair<long long int, long long int> sol = resolver(n);

    // escribir sol
    cout << sol.first << " " << sol.second << '\n';
}

```

Vector parcialmente ordenado

Enunciado:

Los valores de un vector pueden estar más o menos ordenados. En muchos casos nos conviene saber si un vector está *mas o menos* ordenado. Por ejemplo, el algoritmo *quicksort* tiene complejidad cuadrática si el vector está ordenado, mientras que la complejidad del algoritmo de inserción para vectores *casi* ordenados es *casi* lineal. En este problema diremos que un vector está *parcialmente ordenado*, si el valor máximo de su mitad derecha es mayor o igual que todos los valores de la mitad izquierda y el valor mínimo de su mitad izquierda es menor o igual que todos los valores de su mitad derecha. Además tanto la mitad izquierda como la derecha cumplen que están parcialmente ordenados.

Dado un vector de números enteros positivos se pide decidir si está parcialmente ordenado.

Requisitos de implementación.

Indicar la recurrencia utilizada para el cálculo del coste y el coste de la solución obtenida.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de los valores del vector terminados con el valor cero que no forma parte del vector. El último caso de prueba consta únicamente del valor cero, y no debe tratarse.

El número de valores de cada caso de prueba es una potencia de 2.

Salida

Para cada caso de prueba se escribe en una línea *SI* si el vector está parcialmente ordenado y *NO* si no lo está.

Entrada de ejemplo

```
2 6 3 8 0
6 12 8 18 10 15 16 40 0
5 5 5 5 0
2 6 1 8 0
1 3 2 5 3 1 3 4 0
0
```

Salida de ejemplo

```
SI
SI
SI
NO
NO
```

Autor: Isabel Pita.

Solución:

```
//// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

/*
 *   ACLARACIONES:
 *   - N = v.size()
 *
 *   ESPECIFICACIÓN FORMAL
 *
 *   -----
 *   P = { Existe w : 0 <= w : 2^w = N && 0 <= ini < fin <= N }
 *   fun parcial_ordenado(v[ini..fin] de ent, ini : ent, fin : ent) dev <p_ordenado : bool, max : ent, min : ent>
 *   Q = { (p_ordenado = parc_ordenado(v, ini, fin) /\ max = maximo(v, ini, fin) /\ min = minimo(v, ini, fin) }
 *
 *   donde se utilizan los siguientes predicados auxiliares:
 *   - parc_ordenado(v, i, j) = (m = (i + j) / 2 /\ ParaTodo w : i <= w < m : maximo(v, m, j) >= v[w] /\ 
 *   ParaTodo h : m <= h < j : minimo(v, i, m) <= v[h] /\ parc_ordenado(v, i, m) /\ parc_ordenado(v, m, j))
 *   - maximo(v, i, j) = (max w : i <= w < j : v[w])
 *   - minimo(v, i, j) = (min w : i <= w < j : v[w])
 *
 *   RECURRICIA
 *
 *   -----
 *   T(N) = { k_0           si N = 1
 *          { 2 * T(N / 2) + k_1   si N > 1
 *
```

```

/*
 * COSTE
 * -----
 * Si aplicamos el Teorema de la División a partir de la función de recurrencia obtenemos que el coste asintótico
 * de la función recursiva será del orden O(N)
 */
 */

struct solucion {
    bool p_ordenado;
    int max;
    int min;
};

// función recursiva
solucion parcial_ordenado(vector<int> const& v, int ini, int fin) {
    // CASO BASE:
    if (ini == fin - 1) // vector de 1 elemento
        return { true, v[ini], v[ini] };

    // CASO RECURSIVO:
    else {
        int m = (ini + fin) / 2; // punto medio
        // llamadas recursivas:
        solucion izq = parcial_ordenado(v, ini, m);
        solucion dch = parcial_ordenado(v, m, fin);

        bool b = izq.min <= dch.max && dch.max >= izq.max;
        return { b && izq.p_ordenado && dch.p_ordenado, dch.max, izq.min }; /* si dch.max no fuera el máximo de todo el vector (e.d., si el máximo
        fuera izq.max) entonces el vector no sería parcialmente ordenado y darian igual los valores max, min. Esto pasa de manera análoga con izq.min */
    }
}

// función que resuelve el problema
bool resolver(vector<int> const& v) {
    solucion s = parcial_ordenado(v, 0, v.size());
    return s.p_ordenado;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int aux; cin >> aux;
    if (aux == 0)
        return false;
    vector<int> v;
    while (aux != 0) {
        v.push_back(aux);
        cin >> aux;
    }

    bool sol = resolver(v);

    // escribir sol
    if (sol) cout << "SI";
    else cout << "NO";
    cout << '\n';

    return true;
}

```

Números caucásicos

Enunciado:

Un vector de enteros mayores que 0 de longitud 2^n (donde n es un número natural) es *caucásico* si el valor absoluto de la diferencia entre el número de elementos pares de sus mitades es, a lo sumo, 2 y cada mitad también es *caucásica*. Un vector con un elemento es *caucasico*.

Algunos ejemplos:

- $\{2, 4, 6, 8 \parallel 1, 3, 5, 7\}$ No es *caucásico*, porque su primera mitad tiene 4 elementos pares y la segunda 0.
- $\{2, 4, 6, 8 \parallel 2, 8, 5, 10\}$ Es *caucásico*.
- $\{2, 4, 8, 12, 3, 7, 9, 21 \parallel 10, 20, 30, 1, 3, 5, 7, 40\}$ No es *caucásico* ya que la primera mitad no lo es.

Diseña un algoritmo *recursivo* que determine si un vector de longitud 2^n es *caucásico*.

Requisitos de implementación.

Indicar la recurrencia utilizada para el cálculo del coste y el coste de la solución obtenida.

Comentar y explicar la solución propuesta.

El problema se debe resolver utilizando la técnica de divide y vencerás.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba se escribe en dos líneas. En la primera se indica el número de elementos de vector, en la segunda se muestran los valores. La entrada termina con un caso con 0 valores.

El número de valores de cada caso de prueba es 2^n siendo $0 \leq n \leq 20$. Los valores del vector son números enteros positivos.

Salida

Para cada caso de prueba se escribe en una línea *SI* si el vector es caucásico y *NO* si no lo es.

Entrada de ejemplo

```
4
2 6 3 8
8
6 12 8 18 10 15 16 40
8
6 12 8 18 11 15 17 41
16
1 3 2 5 3 1 3 4 4 6 8 2 3 5 7 2
0
```

Salida de ejemplo

```
SI
SI
NO
NO
```

Autor: Isabel Pita.

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ACLARACIONES:
 *   - N = v.size()
 *
 *   ESPECIFICACIÓN FORMAL
 *   -----
 *   P = { Existe n : 0 <= n <= 20 : 2^n = N } \ ParaTodo w : 0 <= w < N : v[w] > 0 } \ 0 <= ini < fin <= N }
 *   fun caucasico(v[ini..fin) de ent, ini : ent, fin : ent) dev <cauc : bool, n_pares : ent>
 *   Q = { cauc = es_caucasico(v, ini, fin) } \ n_pares = num_pares(v, ini, fin) }
 *
 *   donde se utilizan los siguientes predicados auxiliares:
 *   - es_caucasico(v, i, j) = (m = (i + j) / 2 \& |num_pares(v, i, m) - num_pares(v, m, j)| <= 2 ) \& es_caucasico(v, i, m) \& es_caucasico(v, m, j)
 *   - num_pares(v, i, j) = (#w : i <= w < j : v[w] % 2 = 0)
 *
 *   RECURRICIA
 *   -----
 *   T(N) = { k_0           si N = 1
 *           { 2 * T(N / 2) + k_1   si N > 1
 *           
```

```

/*
 * COSTE
 * -----
 * Si aplicamos el Teorema de la División a partir de la función de recurrencia obtenemos que el coste asintótico
 * de la función recursiva será del orden O(N)
 */

// función recursiva
pair<bool, int> caucasico(vector<int> const& v, int ini, int fin) {
    // CASO BASE:
    if (ini == fin - 1) // vector de 1 elemento
        if (v[ini] % 2 == 0) // si es par
            return { true, 1 };
        else
            return { true, 0 };

    // CASO RECURSIVO:
    else {
        int m = (ini + fin) / 2; // punto medio
        // llamadas recursivas:
        pair<bool, int> izq = caucasico(v, ini, m);
        pair<bool, int> dch = caucasico(v, m, fin);

        bool b = abs(izq.second - dch.second) <= 2;
        return { b && izq.first && dch.first, izq.second + dch.second };
    }
}

// función que resuelve el problema
bool resolver(vector<int> const& v) {
    pair<bool, int> sol = caucasico(v, 0, v.size());
    return sol.first;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {
    // leer los datos de la entrada
    int n; cin >> n;
    if (n == 0)
        return false;
    vector<int> v(n);
    for (int i = 0; i < n; ++i)
        cin >> v[i];

    bool sol = resolver(v);

    // escribir sol
    if (sol) cout << "SI";
    else cout << "NO";
    cout << '\n';

    return true;
}

```

Sumar dos números dígito a dígito

Enunciado:

Dados dos números enteros positivos, nos piden obtener el número que resulta de sumar, módulo 10, los dígitos de igual peso de los dos números dados.

El problema debe resolverse dos veces. La primera con una función recursiva no final y la segunda con una función recursiva final.

Requisitos de implementación.

Se deben escribir las ecuaciones recursivas que resuelven el problema en el fichero que se entrega.

Las funciones recursivas deben recibir un número entero, y devolver como valor de retorno el número obtenido al realizar la suma. Se pueden utilizar más parámetros por valor si la solución lo requiere.

$$\begin{array}{r}
 3 \quad 2 \quad 7 \\
 + \quad 4 \quad 5 \\
 \hline
 3 \quad 6 \quad 2
 \end{array}$$

No se deben utilizar parámetros por referencia en ninguna de las funciones implementadas. Se puede devolver más de un valor como retorno de la función si la solución lo requiere.

El problema se debe resolver utilizando un número entero. No se puede utilizar el tipo `std::string`, ni el tipo `std::vector` para resolver el ejercicio.

Entrada

La entrada comienza con el número de casos de prueba. Cada caso se escribe en una línea y consiste en dos números, $0 \leq N_1, N_2 \leq 100.000.000$.

Salida

Para cada caso de prueba se muestra en una línea primero el resultado obtenido con la función recursiva no final y a continuación el resultado de la función recursiva final.

Entrada de ejemplo

```
5
85643 45397
0 365
999 111
9999 111
88 10000
```

Salida de ejemplo

```
20930 20930
365 365
0 0
9000 9000
10088 10088
```

Autor: Isabel Pita

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

/*
 *   ACLARACIONES:
 *   - N = max(n1, n2)
 *   - M = min(n1, n2)
 *
 *   ESPECIFICACIÓN FORMAL (suma)
 *   -----
 *   P = { 0 <= n1, n2 <= 100.000.000 }
 *   fun suma(n1 : ent, n2 : ent) dev sol : ent
 *   Q = { (ParaTodo w : 0 <= w < numDigitos(N) /\ s = digito(n1, w) + digito(n2, w) : (s < 10 --> digito(sol, w) = s) /\ (s >= 10 --> digito(sol, w) = s - 10)) }
 *
 *   donde se utilizan los siguientes predicados auxiliares:
 *   - numDigitos(n) = (min w : n / 10^w = 0 : w)
 *   - digito(n, i) = ((n / 10^i) % 10)
 *
 *   RECURRICIA (suma)
 *   -----
 *   T(M) = { k_0           si N < 10 /\ M = 0
 *           { T(M / 10) + k_1   eoc
 *
 *   COSTE (suma)
 *   -----
 *   Aplicando el Teorema de la División a partir de la función de recurrencia (donde a = 1, b = 10, k = 0) obtenemos que el coste asintótico
 *   en tiempo de la función recursiva es del orden O(log M).
 *
 *   NOTA: Al añadir los casos base (n1 == 0) y (n2 == 0) hemos mejorado un poco el coste de la función, dado que ahora
 *         el coste es de orden logarítmico con respecto al mínimo de (n1, n2), en vez de al máximo. Así, si sumásemos un número muy grande con uno
 *         muy pequeño (por ejemplo, 123456789 + 1) en vez de hacer muchas llamadas recursivas llegaríamos en seguida al caso base.
 *
```

```

/*
 * (sumaFinal)
 * Para la función recursiva final todo esto es igual, con un pequeño cambio en la especificación:
 * P = { 0 <= n1, n2 <= 100.000.000 } \ (Existe w : 0 <= w < numDigitos(N) : potencia = 10^w ) \ \{ (sum = suma que llevamos hasta ahora)? \}
 * fun sumaFinal(n1 : ent, n2 : ent, sum : ent, potencia : ent) dev sol : ent
 */
 */

// función recursiva no final
int suma(int n1, int n2) {

    int sol;

    // CASOS BASE:
    if (n1 < 10 && n2 < 10) { // los dos tienen sólo 1 dígito
        sol = n1 + n2;
        if (sol >= 10) // si hay llevada nos quedamos sólo con el último dígito
            sol -= 10;
    }
    // si alguno de los dos es 0 no hace falta seguir haciendo llamadas recursivas:
    else if (n1 == 0)
        sol = n2;
    else if (n2 == 0)
        sol = n1;

    // CASO RECURSIVO:
    else {
        sol = suma(n1 / 10, n2 / 10);
        int sum = (n1 % 10) + (n2 % 10); // suma últimos dígitos
        if (sum >= 10) // si hay llevada nos quedamos sólo con el último dígito
            sum -= 10;
        sol = sol * 10 + sum;
    }

    return sol;
}

// función recursiva final
int sumaFinal(int n1, int n2, int sum, int potencia) {

    int sol;

    // CASO BASE:
    if (n1 < 10 && n2 < 10) { // los dos tienen sólo 1 dígito
        sol = n1 + n2;
        if (sol >= 10) // si hay llevada nos quedamos sólo con el último dígito
            sol -= 10;
        sol = sol * potencia + sum;
    }
    // si alguno de los dos es 0 no hace falta seguir haciendo llamadas recursivas:
    else if (n1 == 0)
        sol = n2 * potencia + sum;
    else if (n2 == 0)
        sol = n1 * potencia + sum;

    // CASO RECURSIVO:
    else {
        sol = (n1 % 10) + (n2 % 10);
        if (sol >= 10) // si hay llevada nos quedamos sólo con el último dígito
            sol -= 10;
        sol = sumaFinal(n1 / 10, n2 / 10, sol * potencia + sum, potencia * 10);
    }

    return sol;
}

// función que resuelve el problema
pair<int, int> resolver(int n1, int n2) {
    return { suma(n1, n2), sumaFinal(n1, n2, 0, 1) };
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    int n1, n2; cin >> n1 >> n2;

    pair<int, int> sol = resolver(n1, n2);

    // escribir sol
    cout << sol.first << " " << sol.second << '\n';
}

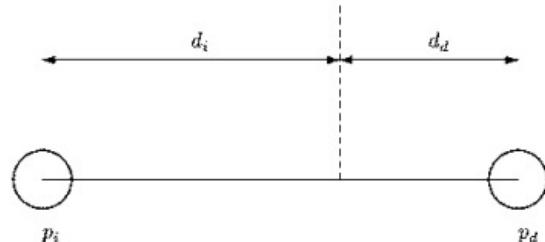
```

Móviles

Enunciado:

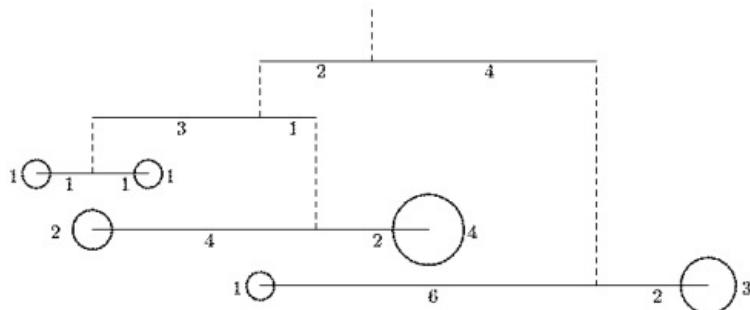
En los años 80 un móvil no era ese dispositivo de comunicación que todos llevamos en el bolsillo, sino una estructura hecha con alambres y cuerdas de las que colgaban figuras coloridas, y que se colocaban sobre las cunas de los bebés para estimularles y entretenérles.

Los móviles que vamos a considerar para este problema están formados por cuerdas, alambres y objetos pesados, y tienen el siguiente aspecto:



La figura representa un móvil muy sencillo con un único alambre colgado de una cuerda, con un objeto pesado a cada lado. El móvil se puede ver como una “balanza” con el punto de apoyo en el sitio donde la cuerda está unida al alambre. Según el principio de la palanca, está en equilibrio si el producto del peso de los objetos por sus distancias al punto de apoyo son iguales. Es decir si consideramos p_i como el peso colgado en el lado izquierdo, p_d el peso del lado derecho, y d_i la distancia desde el peso izquierdo a la cuerda y d_d de la cuerda al peso derecho, podemos decir que el móvil está en equilibrio si $p_i * d_i = p_d * d_d$.

En móviles más complejos, cada peso puede ser sustituido por un *submóvil*. En este caso se considera el peso del submóvil como la suma de los pesos de todos sus objetos, despreciando la cuerda y los alambres. Y consideraremos que está equilibrado si $p_i * d_i = p_d * d_d$ y, además los submóviles de la izquierda y de la derecha están a su vez equilibrados. La siguiente figura muestra un móvil equilibrado.



Se pide diseñar una función eficiente que dado un móvil indique si está en equilibrio o no. El coste en tiempo de esta función debe ser *lineal* (con respecto al número de objetos pesados del móvil).

Entrada

La entrada comienza con una línea con el número de casos de prueba que vienen a continuación. Cada caso de prueba es un móvil, descrito con una o varias líneas, cada una de ellas conteniendo cuatro números enteros positivos, separados por un único espacio. Esos cuatro enteros representan las distancias de los extremos al punto de apoyo, así como sus pesos, en el orden p_i, d_i, p_d, d_d . Si p_i o p_d (alguno de los pesos) es 0, significa que en el extremo habrá colgado un submóvil, que estará descrito a continuación. Si un móvil tiene un submóvil en cada lado, primero se describirá el submóvil izquierdo.

Salida

El programa indicará **SI** si el móvil que representa está en equilibrio, y **NO** en otro caso.

Entrada de ejemplo

```
2
0 2 0 4
0 3 0 1
1 1 1 1
2 4 4 2
1 6 3 2
0 1 3 4
2 3 3 2
```

Salida de ejemplo

```
SI
NO
```

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

/*
 * Aclaraciones
 *
 * Este problema está visto como un árbol, en el que cada nodo es un par <peso, distancia> representando
 * un objeto (o submóvil) del móvil que estamos estudiando. La raíz del árbol es el par <0, 0>, de donde parten
 * los dos primeros hijos, que a su vez pueden tener hijos propios si el hijo en cuestión es un submóvil (e.d.,
 * si su peso = 0). Cada nodo tiene siempre 2 hijos, y las hojas del árbol serán aquellos objetos que no sean
 * submóviles (y que por tanto no tienen más hijos)
 * N = tamaño del árbol, e.d., el número de nodos del árbol. Esto es, el número de objetos/submóviles que tiene
 * el móvil
 *
 * Especificación formal
 *
 * P = { m.p >= 0 /\ m.d > 0 }
 * fun equilibrado(<m.p, m.d> : <ent, ent>) dev <eq : bool, suma : ent>
 * Q = { (m.p != 0 --> (eq = true /\ suma = m.p)) /\ (m.p = 0 --> eq_rec(eq, suma)) }
 *
 * donde se utiliza el siguiente predicado auxiliar:
 * - eq_rec(eq, suma) = ((Existe n1.p, n1.d, n2.p, n2.d : n1.p * n1.d = n2.p * n2.d) /\ (a1 = equilibrado(n1.p, n1.d)) /\ (a2 = equilibrado(n2.p, n2.d)) /\
 *                         /\ (eq = a1.eq /\ a2.eq) /\ (suma = a1.suma + a2.suma))
 *
 * Recurrencia
 *
 * Para escribir la recurrencia es necesario distinguir casos en función de la distribución del árbol. Para ello,
 * estudiaremos los dos casos extremos:
 * 1) Aquel en el que todos los hijos izquierdos (o derechos) son submóviles (excepto el último, obviamente). Esto es,
 *    numNodosHijoIzq = N aproximadamente
 * 2) Aquel en el que los nodos están distribuidos uniformemente en el árbol, e.d., el que cumpla numNodosHijoIzq = numNodosHijoDch
 *
 * De esta forma, la recurrencia quedaría para cada caso de la siguiente forma:
 * 1) T(N) = { k_0                                si N = 1
 *             { T(N - 1) + T(1) + k_1 = T(N - 1) + k_2 + k_1 = T(N - 1) + k_3      si N > 1
 * 2) T(N) = { k_0                                si N = 1
 *             { T(N / 2) + T(N / 2) + k_1 = 2T(N / 2) + k_1      si N > 1
 *
 * Coste
 *
 * 1) Para la primera función de recurrencia, si aplicamos el Teorema de la Resta (donde a = 1, b = 1, k = 0), obtenemos que el coste
 *    asintótico en tiempo de la función recursiva es del orden O(N).
 * 2) Para la segunda función de recurrencia, si aplicamos el Teorema de la División (donde a = 2, b = 2, k = 0), obtenemos que el
 *    coste asintótico en tiempo de la función recursiva es del orden O(N).
 * Por lo tanto, podemos concluir que para cualquier distribución del árbol el coste será lineal, lo cual tiene sentido, dado que tenemos
 * que recorrer todos los nodos del árbol (e.d., N nodos) y "pasamos" por cada uno de ellos 1 única vez.
 * NOTA: "Pasamos" entre comillas porque realmente el árbol no existe en un principio, sino que lo vamos recorriendo según lo vamos
 *       construyendo (realmente tampoco construimos la estructura de datos, simplemente vamos leyendo los nodos de la entrada).
 */

struct objeto {
    int p; // peso
    int d; // distancia
};
```

```

// función recursiva
pair<bool, int> equilibrado(objeto o) {
    // CASO BASE:
    if (o.p != 0)
        return { true, o.p };

    // CASO RECURSIVO:
    else {
        // leemos la entrada:
        int p_i, d_i; cin >> p_i >> d_i;
        int p_d, d_d; cin >> p_d >> d_d;

        // llamadas recursivas:
        pair<bool, int> izq = equilibrado({ p_i, d_i });
        pair<bool, int> dch = equilibrado({ p_d, d_d });

        return { (izq.first && dch.first && (izq.second * d_i == dch.second * d_d)), izq.second + dch.second };
    }
}

// función que resuelve el problema
bool resolver() {
    return equilibrado({ 0, 0 }).first;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {
    bool sol = resolver();

    // escribir sol
    if (sol) cout << "SI";
    else cout << "NO";
    cout << '\n';
}

```

Fuga de la prisión

Enunciado:

Año 3016, sistema interestelar XG23. En la carcel de alta seguridad C78 se ha detectado la fuga de un preso del pabellón Z4.

Cada preso del pabellón se identifica mediante una letra del alfabeto latino moderno de 26 letras. Sabiendo que el primer preso de este pabellón tiene la letra x_1 y el último es el x_2 y que todos ellos son consecutivos, encuentra la letra del preso fugado lo más rápido posible para evitar que pueda robar una nave y salir del sistema interestalar.

Requisitos de implementación.

Indicar el coste de la solución obtenida.

La función que resuelve el problema debe recibir las letras de los presos en un vector y calcular el que falta. La implementación debe ser recursiva.

Dada dos variables de tipo *char* el resultado de restar su valor es la diferencia entre los códigos ASCII de los caracteres. Por ejemplo si la variable x_1 tiene el valor '*c*' y la variable x_2 el valor '*f*', el resultado de la operación $x_2 - x_1$ es 3.



Entrada

La entrada comienza con el número de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica la letra del primer y último presidiario del pabellón. En la segunda se indican las letras de los presidiarios presentes en el recuento, ordenados en orden ascendente.

Cada pabellón tiene al menos un preso distinto del que se ha fugado.

Salida

Para cada caso de prueba se escribe en una línea la letra del preso fugado.

Entrada de ejemplo

```
12
a e
a b c e
c h
d e f g h
c h
c e f g h
c h
c d f g h
c h
c d e g h
c h
c d e f h
c h
c d e f g
d h
e f g h
d h
d f g h
d h
d e g h
d h
d e f h
d h
d e f g
```

Salida de ejemplo

```
d
c
d
e
f
g
h
d
e
f
g
h
```

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   Aclaraciones:
 *   - N = presos.size()
 *
 *   Especificación formal
 *
 *   P = { N > 0 /\ N = u - p /\ ordenado(presos, N, p, u) /\ no_repetidos(presos, N) /\ falta_uno(presos, N, p, u) }
 *   fun preso_fugado(presos[0..N] de char, p : char, u : char, ini : ent, fin : ent) dev fug : char
 *   Q = { (p <= fug <= u) /\ !pertenece(presos, N, fug) }
 *
 *   donde se utilizan los siguientes predicados auxiliares:
 *   - ordenado(v, n, p, u) = (ParaTodo i,j : 0 <= i <= j < N : p <= v[i] <= v[j] <= u)
 *   - no_repetidos(v, n) = (!Existe i,j : 0 <= i < j < N : v[i] = v[j])
 *   - falta_uno(v, n, p, u) = ((Existe f : p <= f <= u : !pertenece(v, n, f)) /\ (ParaTodo i : p <= i <= u /\ i != f : pertenece(v, n, i)))
 *   - pertenece(v, n, k) = (Existe w : 0 <= w < n : v[w] = k)
 *
```

```

/*
 * RECURRICIA
 *
 * -----
 * T(N) = { k_0           si N = 1
 *          { T(N / 2) + k_1   si N > 1
 *
 * COSTE
 * -----
 * Si aplicamos el Teorema de la División a partir de la función de recurrencia (donde a = 1, b = 2, k = 0)
 * obtenemos que el coste asintótico en tiempo de la función recursiva es del orden O(log N)
 *
 */

```

```

// función recursiva
char preso_fugado(vector<char> const& presos, char p, char u, int ini, int fin) {

    // CASO BASE:
    if (ini == fin - 1) // solo 1 preso (aparte del fugado)
        if (presos[ini] == p)
            return u;
        else return p;

    // CASO RECURSIVO:
    else {
        int m = (ini + fin) / 2;
        char medio = (p + u) / 2;
        if (presos[m] > medio)
            return preso_fugado(presos, p, medio, ini, m);
        else
            return preso_fugado(presos, medio, u, m, fin);
    }
}

// función que resuelve el problema
char resolver(vector<char> const& presos, char prim, char ult) {
    return preso_fugado(presos, prim, ult, 0, presos.size());
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    char prim; cin >> prim;
    char ult; cin >> ult;
    vector<char> presos(ult - prim);
    for (int i = 0; i < presos.size(); ++i)
        cin >> presos[i];

    char sol = resolver(presos, prim, ult);

    // escribir sol
    cout << sol << '\n';
}

```

El juego del bongo

Enunciado:

En la residencia de mis tíos, se ha puesto de moda un nuevo juego al que llaman el *bongo*. Cansados de tachar el número que cantaba el cajero sobre su tarjeta de bingo, los ancianos han optado por tachar el número del cartón que coincide con el número cantado por el cajero mas su posición en el cartón (la primera posición del cartón es la cero). De esta forma ejercitan su capacidad de cálculo.

Así, por ejemplo, si en el cartón están los números 2 4 7 9 12 15 22 y el número cantado es el 6, al anciano tachará el número 9, ya que su posición, la 3, más el número cantado coinciden con el valor. Si no encuentra ningún número que le coincida no tachará ninguno y esperará al siguiente número. Utilizan únicamente cartones con los números ordenados de forma creciente y para evitar que una persona pueda tachar dos números en una tirada, los cartones no tienen ninguna pareja de números consecutivos.

Hoy he estado jugando con ellos, pero me falta práctica y perdía algunos números. He decidido hacer un programa que dados los valores del cartón y el número que canta el cajero me enseñe el número que debo tachar o si no debo hacer nada, para poder estar a su nivel la próxima vez que juegue.



Requisitos de implementación.

Indicar el coste de la solución obtenida.

La función que resuelve el problema debe ser recursiva.

Entrada

La entrada comienza con el número de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica la cantidad de números que hay en el cartón y el número cantado por el cajero. En la segunda línea se indican los valores del cartón separados por blancos.

Los cartones tienen al menos un elemento.

Salida

Para cada caso de prueba se escriben en una línea el número que se debe tachar o la palabra *NO* si no hay ninguno en su sitio.

Entrada de ejemplo

```
4
7 8
3 6 8 11 14 17 20
6 2
0 3 5 7 9 11
4 5
0 4 7 12
5 4
1 3 7 9 12
```

Salida de ejemplo

```
11
3
7
NO
```

Autor: Isabel Pita.

Solución:

```
//// Beatriz Espinar Aragón
//// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   DECLARACIONES:
 *   - N = carton.size()
 *
 *   ESPECIFICACIÓN FORMAL
 *   -----
 *   P = { N > 0 } \ ordenado(carton, N) /\ 0 <= ini <= fin <= N }
 *   fun tachar(carton[0..N) de ent, k : ent, ini : ent, fin : ent) dev sol : ent
 *   Q = { (!hayQueTachar(carton, ini, fin, k) --> sol = -1) /\ (hayQueTachar(carton, ini, fin, k) --> (sol = carton[i] /\ ini <= i < fin /\ carton[i] = k + i)) }
 *
 *   donde se utiliza el siguiente predicado auxiliar:
 *   - ordenado(v, n) = (ParaTodo w : 0 <= w < n - 1 : v[w] < v[w + 1])
 *   - hayQueTachar(v, i, j, k) = (Existe w : i <= w < j : v[w] = k + w)
 *
 *   RECURRICIA
 *   -----
 *   T(N) = { k_0           si N = 0
 *           { T(N / 2) + k_1   si N > 0       (en el caso mejor será sólo k_1 si carton[m] = k + m)
 *
 *   COSTE
 *   -----
 *   Si aplicamos el Teorema de la División a partir de la función de recurrencia (donde a = 1, b = 2, k = 0) obtenemos
 *   que el coste asintótico en tiempo de la función recursiva es del orden O(log N)
 */


```

```

// función recursiva:
int tchar(vector<int> const& carton, int k, int ini, int fin) {
    // CASO BASE:
    if (ini == fin) // vector vacío
        return -1;

    // CASO RECURSIVO:
    else {
        int m = (ini + fin) / 2; // mitad
        if (carton[m] == k + m)
            return carton[m];
        else if (carton[m] > k + m)
            return tchar(carton, k, ini, m);
        else
            return tchar(carton, k, m + 1, fin);
    }
}

// función que resuelve el problema
int resolver(vector<int> const& carton, int k) {
    return tchar(carton, k, 0, carton.size());
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    int k; cin >> k;
    vector<int> carton(n);
    for (int i = 0; i < n; ++i)
        cin >> carton[i];

    int sol = resolver(carton, k);

    // escribir sol
    if (sol == -1) cout << "NO";
    else cout << sol;
    cout << '\n';
}

```

Buscar el elemento perdido

Enunciado:

Se tienen dos vectores, a y b , de enteros ordenados y distintos entre sí con n y $n-1$ elementos respectivamente. Los elementos de b son los mismos que tiene a excepto uno que falta. Se pide implementar un algoritmo recursivo eficiente que encuentre ese valor que falta. Se debe indicar la recurrencia y el coste asintótico en el caso peor del algoritmo. (Examen junio 2017)

Requisitos de implementación.

Se debe implementar una función recursiva que reciba los dos vectores y devuelva el valor que falta. Pueden utilizarse más parámetros de entrada si hacen falta.

Se valorará la eficiencia de la solución, el uso eficiente de variables y parámetros, los comentarios sobre el algoritmo y sobre su coste.

Entrada

La entrada comienza con el número de casos a tratar. Cada caso consta de 3 líneas, en la primera se da el número de elementos del primer vector, en la segunda los elementos del primer vector y en la tercera los elementos del segundo vector.

Los valores de los vectores son números enteros.

Salida

Para cada caso de prueba se escribe en una línea el número que falta.

Entrada de ejemplo

```
5
5
10 20 30 40 50
10 30 40 50
5
10 20 30 40 50
10 20 30 40
5
10 20 30 40 50
20 30 40 50
1
6

2
1 8
1
```

Salida de ejemplo

```
20
50
10
6
8
```

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *  ACLARACIONES:
 *  - N = a.size()
 *  - M = b.size()
 *
 *  ESPECIFICACIÓN FORMAL
 *  -----
 *  P = { N > 0 } /\ M = N - 1 /\ ordenado_sin_repetidos(a, N) /\ copia_ordenada_sin_repetidos(b, M, a, N) /\ falta_uno(b, M, a[0], a[N - 1])
 *  fun elemento_perdido(a[0..N] de ent, b[0..M] de ent, ini : ent, fin : ent) dev perdido : ent
 *  Q = { a[0] <= perdido <= a[N - 1] } /\ !pertenece(b, M, perdido) /\ pertenece(a, N, perdido)
 *
 *  donde se utilizan los siguientes predicados auxiliares:
 *  - ordenado(v, n) = (ParaTodo w : 0 <= w < n - 1 : v[w] < v[w + 1])
 *  - copia_ordenada_sin_repetidos(b, m, a, n) = (ParaTodo w : 0 <= w < m - 1 : a[0] <= b[w] < b[w + 1] <= a[n - 1])
 *  - falta_uno(b, m, p, u) = ((Existe f : p <= f <= u : !pertenece(b, m, f)) /\ (ParaTodo i : p <= i <= u /\ i != f : pertenece(b, m, i)))
 *  - pertenece(v, n, k) = (Existe w : 0 <= w < n : v[w] = k)
 *
 *  RECURRICIA
 *  -----
 *  T(M) = { k_0           si M = 0
 *          { T(M / 2) + k_1   si M > 0
 *
 *  COSTE
 *  -----
 *  Si aplicamos el Teorema de la División a partir de la función de recurrencia (donde a = 1, b = 2, k = 0), obtenemos que
 *  el coste asintótico en tiempo de la función recursiva es del orden O(log M), que a su vez está en el orden O(log N), dado
 *  que M = N - 1
 */

```

```

// función recursiva
int elemento_perdido(vector<int> const& a, vector<int> const& b, int ini, int fin) {
    // CASO BASE:
    if (ini >= fin) // b vacío
        return a[ini];

    // CASO RECURSIVO:
    else {
        int m = (ini + fin) / 2;
        if (b[m] > a[m])
            return elemento_perdido(a, b, ini, m);
        else
            return elemento_perdido(a, b, m + 1, fin);
    }
}

// función que resuelve el problema
int resolver(vector<int> const& a, vector<int> const& b) {
    return elemento_perdido(a, b, 0, b.size());
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // Leer los datos de la entrada
    int n; cin >> n;
    vector<int> a(n), b(n - 1);
    for (int i = 0; i < n; ++i)
        cin >> a[i];
    for (int j = 0; j < n - 1; ++j)
        cin >> b[j];

    int sol = resolver(a, b);

    // Mostrar la solución
    cout << sol << '\n';
}

```

Obtener el mínimo de un vector

Enunciado:

Consideramos un vector $V[N]$ de números enteros, cuyo valores se han obtenido aplicando una rotación sobre un vector ordenado en orden estrictamente decreciente. Implementa un algoritmo que calcule el mínimo del vector con una complejidad $O(\log(n))$. El número de elementos sobre los que se aplica la rotación para obtener el vector de entrada es un valor entre 0 y N y no se conoce.

Por ejemplo, un posible vector de entrada sería el vector 70 55 13 4 100 80 obtenido desplazando los dos primeros elementos del vector 100 80 70 55 13 4 al final del mismo.

Obtenido del examen final de septiembre de 2014.

Requisitos de implementación.

Se debe implementar una función recursiva (`resolver`) que dado el vector, con los datos de entrada ya leídos, obtenga el mínimo en tiempo logarítmico respecto al número de elementos del vector. Se pueden utilizar más parámetros si se considera necesario.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de dos líneas. En la primera se indica el número de elementos del vector y en la segunda los valores del vector.

Salida

Para cada caso de prueba se escribe en una línea diferente el mínimo del vector.

Entrada de ejemplo

```
4
8 6 4 2
4
4
2 8 6 4
4
4 2 8 6
4
6 4 2 8
5
8 5 3 1 10
5
5 3 1 10 8
6
70 55 13 4 100 80
```

Salida de ejemplo

```
2
2
2
2
1
1
4
```

Autor: Isabel Pita.

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ACLARACIONES:
 *   - N = v.size()
 *
 *   ESPECIFICACIÓN FORMAL
 *   -----
 *   P = { N > 0 } \ (rotado_decreciente(v, N) \vee decreciente(v, 0, N)) /\ 0 <= ini < fin <= N }
 *   fun minimo(v[0..N]) de ent, ini : ent, fin : ent) dev min : ent
 *   Q = { min = min w : 0 <= w < N : v[w] }

 *   donde se utiliza el siguiente predicado auxiliar:
 *   - rotado_decreciente(v, n) = (Existe p : 0 <= p < N - 1 : decreciente(v, 0, p) /\ decreciente(v, p + 1, n) /\ v[p] < v[p + 1])
 *   - decreciente(v, i, j) = (ParaTodo w : i <= w < j - 1 : v[w] > v[w + 1])

 *   RECURRENCIA
 *   -----
 *   T(N) = { k_0           si N = 1 \vee rotacion = 0
 *           { T(N / 2) + k_1   eoc
 *
 *   COSTE
 *   -----
 *   Si aplicamos el Teorema de la División a partir de la función de recurrencia (donde a = 1, b = 2, k = 0) obtenemos que
 *   el coste asintótico en tiempo de la función recursiva es del orden O(log N)
 *
 */

```

```

// función recursiva
int minimo(vector<int> const& v, int ini, int fin) {
    // CASO BASE:
    if (ini == fin - 1) // sólo 1 elemento
        return v[ini];
    // Podemos añadir otro caso base para terminar antes:
    else if (v[ini] > v[fin - 1]) // si rotación = 0
        return v[fin - 1];

    // CASO RECURSIVO:
    else {
        int m = (ini + fin) / 2;
        if (v[m] > v[ini])
            return minimo(v, ini, m);
        else
            return minimo(v, m, fin);
    }
}

// función que resuelve el problema
int resolver(vector<int> const& v) {
    return minimo(v, 0, v.size());
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {
    // leer los datos de la entrada
    int N; cin >> N;
    if (!std::cin)
        return false;
    vector<int> v(N);
    for (int i = 0; i < N; ++i)
        cin >> v[i];

    int sol = resolver(v);

    // escribir sol
    cout << sol << '\n';
    return true;
}

```

Degrado de una imagen

Enunciado:

En diseño gráfico un degradado es un rango de colores ordenados con la intención de dar visualmente una transición suave y progresiva entre dos o más colores. En el software informático los degradados se consiguen reduciendo progresivamente el porcentaje del primer color mientras se aumenta el porcentaje del segundo. Si reducimos el porcentaje de blanco en un color y aumentamos el porcentaje de negro obtendremos colores más oscuros.

Vamos a representar los colores claros con valores pequeños (poca proporción de color negro) y los colores oscuros con valores altos (mayor proporción de color negro). Dada una imagen, representada mediante una matriz de números enteros, nos piden comprobar si es aproximadamente un degradado. En el problema consideraremos que la matriz es un degradado si cada fila cumple que la suma de los colores de la mitad izquierda de la imagen es menor que la suma de los valores de la mitad derecha. Además, para conseguir un degradado uniforme cada mitad debe ser a su vez un degradado.



Requisitos de implementación.

Indicar la recurrencia utilizada para el cálculo del coste y el coste de la solución obtenida.

La solución obtenida debe emplear la técnica de divide y vencerás.

La función que resuelve el problema debe recibir los datos de una fila en un vector y comprobar si es un degradado. Se pueden utilizar parámetros auxiliares para tratar la recursión.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba representa una matriz. Comienza con una línea que indica las dimensiones de la matriz $n * m$ seguida de n líneas con m valores, que representan las filas de la matriz.

El número de valores en cada fila de la matriz m es una potencia de 2.

Salida

Para cada caso de prueba se escribe en una línea *SI* si la matriz es un degradado y *NO* si no lo es.

Entrada de ejemplo

```
3 4
1 3 2 6
2 5 4 11
2 3 3 10
3 8
2 6 7 12 1 8 15 22
2 5 2 6 2 3 3 4
1 2 3 4 5 6 7 8
2 8
2 5 2 16 2 8 16 50
1 3 4 5 2 8 14 15
1 4
1 3 2 3
```

Salida de ejemplo

```
SI
NO
SI
SI
```

Autor: Isabel Pita.

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ACLARACIONES:
 *   - M = fila.size()      (= número de columnas de la matriz)
 *
 *   ESPECIFICACIÓN FORMAL
 *   -----
 *   P = { (Existe i : 0 <= i : M = 2^i) /\ 0 <= ini < fin <= M }
 *   fun degradado(fila[0..M] de ent, ini : ent, fin : ent) dev <deg : bool, sum : ent>
 *   Q = { m = (ini + fin / 2) /\ deg = (suma(fila, ini, m) < suma(fila, ini, fin)) /\ degradado(fila, ini, m) /\ degradado(fila, m, fin) } /\ sum = suma(fila, ini, fin) }
 *
 *   donde se utiliza el siguiente predicado auxiliar:
 *   - suma(v, i, j) = (SUM w : i <= w < j : v[w])
 *
 *   RECURRENCIA
 *   -----
 *   T(M) = { k_0           si M = 1
 *           { 2T(M / 2) + k_1   si M > 1
 *
 *   COSTE
 *   -----
 *   Si aplicamos el Teorema de la División a partir de la función de recurrencia (donde a = 2, b = 2, k = 0) obtenemos que el coste asintótico
 *   en tiempo de la función recursiva es del orden O(M).
 *   Este coste es el de ver si una fila es degradada. Dado que para el algoritmo completo hay que mirar todas las filas (bucle de la función resolver), el coste
 *   total será del orden O(N * M)
 */


```

```

// función recursiva
pair<bool, int> degradado(vector<int> const& fila, int ini, int fin) {
    // CASO BASE:
    if (ini == fin - 1) // 1 elemento
        return { true, fila[ini] };

    // CASO RECURSIVO:
    else {
        int m = (ini + fin) / 2;

        // llamadas recursivas:
        pair<bool, int> izq = degradado(fila, ini, m);
        pair<bool, int> dch = degradado(fila, m, fin);

        return { izq.first && dch.first && izq.second < dch.second, izq.second + dch.second };
    }
}

// función que resuelve el problema
bool resolver(vector<vector<int>> const& imagen, int N, int M) {
    bool deg = true;
    int cont = 0;

    while (deg && cont < N) { // coste O(N)
        deg = degradado(imagen[cont], 0, M).first;
        ++cont;
    }

    return deg;
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {
    // leer los datos de la entrada
    int n; cin >> n;
    if (!std::cin)
        return false;
    int m; cin >> m;
    vector<vector<int>> imagen(n, vector<int>(m));
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < m; ++j)
            cin >> imagen[i][j];

    bool sol = resolver(imagen, n, m);

    // escribir sol
    if (sol) cout << "SI";
    else cout << "NO";
    cout << '\n';

    return true;
}

```

Battlestar Galactica

Enunciado:

En un lejano lugar del universo, los *Cylon* han lanzado un ataque imprevisto contra las doce colonias de Kobol. Todo lo que queda de la humanidad es una nave de combate la *Battlestar Galactica* junto con un grupo de naves espaciales civiles. Juntos emprenden un viaje en busca de la decimotercera colonia: la Tierra, cuya ubicación es desconocida. El comandante *Adama* dirige la expedición desde la nave de combate y el resto de las naves se sitúan en línea detrás de la nave principal. El orden en que lo hacen es importante para poder defenderse mejor de los continuos ataques de los *Cylon*.

Después del último ataque las naves han quedado desordenadas y es preciso que recuperen el orden correcto. Sin embargo, el comandante sabe que los *Cylon* preparan un nuevo ataque para dentro de T unidades de tiempo. Si intercambiar el lugar de dos naves consecutivas lleva una unidad de tiempo, calcula si será posible reestructurar la fila antes del ataque, en caso contrario, los pilotos *Apolo* y *Starbuck* tendrán que realizar una salida contra los *Cylon* para intentar retrasar el ataque.

Requisitos de implementación.

El problema se debe resolver de forma recursiva. Indicar el coste obtenido y la recurrencia .

La función que resuelve el problema debe recibir los datos en un vector y devolver el número de unidades de tiempo necesarias para reestructurar la fila de naves.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso consta de dos líneas, en la primera se indica el número de naves que viajan bajo la protección de la *Battlestar Galactica*. En la segunda línea se da la posición en que ha quedado cada nave después del ataque de los *Cylon*. Por ejemplo, si la entrada es 5 2 1 3 4, esto significa que la nave 5 (la que debería viajar en quinto lugar) está ahora en primera posición, la nave que debería estar en la segunda posición esta bien colocada, la nave 1, que debería estar en la primera posición, ocupa la tercera, y las naves que deberían estar en tercera y cuarta posición, han quedado en las posiciones 4 y 5.

Salida

Para cada caso de prueba se escribe en una línea el número de unidades de tiempo necesarias para reestructurar la fila.

Entrada de ejemplo

```
3
3 2 1
6
4 3 2 1 6 5
```

Salida de ejemplo

```
3
7
```

Autor: Isabel Pita.

Solución:

```
//// Beatriz Espinar Aragón
//// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

/*
 *   ACLARACIONES:
 *   - N = naves.size()
 *
 *   ESPECIFICACIÓN FORMAL
 *   -----
 *   P = { N >= 0 } \ ParaTodo w : 0 <= w < N : 0 < naves[w] <= N }
 *   fun reestructurar(naves[0..N] de ent, ini : ent, fin : ent) dev ordenado[ini..fin] de ent, intercambios : ent
 *   Q = { ordenado = perm(naves, ini, fin) /\ ord(ordenado, ini, fin) /\ intercambios = {...} }
 *
 *   donde se utilian los siguientes predicados auxiliares:
 *   - perm(v, i, f) = "permutación de los elementos del vector v entre las posiciones [i..f]"
 *   - ord(v, i, f) = (ParaTodo w : i <= w < f - 1 : v[w + 1] - v[w] = 1)
 *
 *   RECURRENCIA
 *   -----
 *   T(N) = { k_0                                si N <= 1
 *           { 2 * T(N / 2) + k_1 * N + k_2 * N / 2 + k_3      si N > 1
 *   *
```

```

/*
 * Esto es equivalente a:
 * T(N) = { k_0                      si N <= 1
 *          { 2 * T(N / 2) + k_4 * N    si N > 1
 *
 * COSTE
 * -----
 * Si aplicamos el Teorema de la División a partir de la función de recurrencia (donde a = 2, b = 2, k = 1) obtenemos
 * que el coste asintótico en tiempo de la función recursiva es del orden O(N log N).
 */
*/

```

```
using solucion = pair<int, vector<int>>;
```

```

vector<int> mezcla_ordenada(vector<int> const& izq, vector<int> const& dch) {
    vector<int> ord(izq.size() + dch.size());

    int cont = 0, cont_izq = 0, cont_dch = 0;
    while (cont_izq < izq.size() && cont_dch < dch.size()) {
        if (izq[cont_izq] < dch[cont_dch]) {
            ord[cont] = izq[cont_izq];
            ++cont_izq;
        } else {
            ord[cont] = dch[cont_dch];
            ++cont_dch;
        }
        ++cont;
    }

    // añadimos los que faltan:
    while (cont_izq < izq.size()) {
        ord[cont] = izq[cont_izq];
        ++cont_izq;
        ++cont;
    }

    while (cont_dch < dch.size()) {
        ord[cont] = dch[cont_dch];
        ++cont_dch;
        ++cont;
    }

    return ord;
}

```

```

// función recursiva
solucion reestructurar(vector<int> const& naves, int ini, int fin) {

    int N = fin - ini;
    vector<int> ordenado(N);
    int intercambios;

    // CASO BASE:
    if (ini == fin) // vector vacío
        return { 0, ordenado };

    if (ini == fin - 1) { // sólo 1 nave
        ordenado[0] = naves[ini];
        intercambios = 0;
        return { intercambios, ordenado };
    }

    // CASO RECURSIVO:
    else {

        int m = (ini + fin) / 2;
        // llamadas recursivas:
        solucion izq = reestructurar(naves, ini, m);
        solucion dch = reestructurar(naves, m, fin);

        // ordenamos mezclando:
        ordenado = mezcla_ordenada(izq.second, dch.second); // O(N)
    }
}

```

```

    // calculamos intercambios:
    intercambios = izq.first + dch.first;
    int cont = 0, cont_dch = 0;
    bool ya_colocados = false;
    while (cont < ordenado.size() && cont_dch < dch.second.size() && !ya_colocados) { // da como máx N / 2 vueltas
        if (ordenado[cont] < dch.second[cont_dch])
            ++cont;
        else {
            if (m + cont_dch == cont) // si está en la pos correcta
                ya_colocados = true;
            else
                intercambios += m + cont_dch - (cont + ini);
            ++cont;
            ++cont_dch;
        }
    }

    return { intercambios, ordenado };
}

// función que resuelve el problema
int resolver(vector<int> const& naves) {
    return reestructurar(naves, 0, naves.size()).first;
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    if (!std::cin)
        return false;
    vector<int> naves(n);
    for (int i = 0; i < n; ++i)
        cin >> naves[i];

    int sol = resolver(naves);

    // escribir sol
    cout << sol << '\n';
    return true;
}

```

Las N reinas

Enunciado:

Descripción

En este ejercicio hay que implementar una versión *genérica* del problema de las reinas. En este caso queremos conocer el **número de maneras diferentes** en las que se pueden colocar N reinas en un tablero de ajedrez de tamaño $N \times N$ sin que se ataquen entre ellas.

Para solucionar este problema debes seguir el paradigma de **vuelta atrás** con la técnica de **marcado/desmarcado** para comprobar lo más rápido posible cuándo una solución candidata es solución real y cuándo una solución parcial es completable.

Entrada

La entrada comenzará con una linea conteniendo un número natural M que indica la cantidad casos de prueba que vamos a considerar. Cada caso de prueba constará de un número $1 \leq N \leq 14$ con el número de reinas a colocar en un tablero de $N \times N$.

Salida

Por cada caso de prueba se mostrará una línea con el número total de maneras diferentes de colocar N reinas en un tablero de tamaño $N \times N$. El número de soluciones siempre se podrá almacenar sin desbordamiento en una variable de tipo `int`.

Ejemplo de entrada

```
3  
2  
3  
4
```

Ejemplo de salida

```
0  
0  
2
```

Solución:

```
// Beatriz Espinar Aragón  
// F19  
  
#include <iostream>  
#include <iomanip>  
#include <fstream>  
#include <vector>  
using namespace std;  
  
/*  
 *   ESPACIO DE BÚSQUEDA  
 *   -----  
 *   El espacio de búsqueda es un árbol en el que la solución es una tupla  $(X_0, \dots, X_{n-1})$ , donde  $n =$  número de reinas  
 *   y ParaTodo  $k : 0 \leq k < n : (X_k = \text{sol}[k]) =$  fila en la que colocamos la reina de la columna  $k$   $\wedge 0 \leq X_k < n$   
 *  
 *   MARCADORES  
 *   -----  
 *   Los marcadores empleados para mejorar la eficiencia de las funciones es_solucion() y es_completable() son los siguientes:  
 *   - filas: ParaTodo  $i : 0 \leq i < n : \text{filas}[i] =$  la fila  $i$  está ocupada  
 *   - d_asc: ParaTodo  $i : 0 \leq i < 2 * n - 1 : d_{\text{asc}}[i] =$  la diagonal ascendente  $i$  está ocupada  
 *   - d_desc: ParaTodo  $i : 0 \leq i < 2 * n - 1 : d_{\text{desc}}[i] =$  la diagonal descendente  $i$  está ocupada*  
 */  
  
// calcula la diagonal ascendente:  
int diag_asc(const int x, const int y) {  
    return (x + y);  
}  
  
// calcula la diagonal descendente:  
int diag_desc(const int x, const int y, const int n) {  
    return (x - y + (n - 1));  
}  
  
// comprueba si la casilla que estamos mirando está amenazada por alguna reina:  
bool no_ataca(vector<int> const& sol, const int k, const int n, vector<bool> const& filas, vector<bool> const& d_asc, vector<bool> const& d_desc) {  
    return (!filas[sol[k]] && !d_asc[diag_asc(k, sol[k])] && !d_desc[diag_desc(k, sol[k], n)]);  
}  
  
// comprueba si lo que llevamos hasta ahora es solución:  
bool es_solucion(vector<int> const& sol, const int k, const int n, vector<bool> const& filas, vector<bool> const& d_asc, vector<bool> const& d_desc) {  
    return (k == n - 1) && no_ataca(sol, k, n, filas, d_asc, d_desc);  
}  
  
// comprueba si lo que llevamos hasta ahora puede llegar a ser solución:  
bool es_completable(vector<int> const& sol, const int k, const int n, vector<bool> const& filas, vector<bool> const& d_asc, vector<bool> const& d_desc) {  
    return (k < n - 1) && no_ataca(sol, k, n, filas, d_asc, d_desc);  
}  
  
// aumenta el número de soluciones encontradas:  
void procesar_solucion(int& cuantas_sol) {  
    ++cuantas_sol;  
}
```

```

// algoritmo vuelta-atrás:
void n_reinas(vector<int>& sol, const int k, const int n, vector<bool>& filas, vector<bool>& d_asc, vector<bool>& d_desc, int& cuantas_sol) {
    for (int i = 0; i < n; ++i) {
        sol[k] = i;
        if (es_solución(sol, k, n, filas, d_asc, d_desc))
            procesar_solución(cuantas_sol);
        else if (es_completetable(sol, k, n, filas, d_asc, d_desc)) {
            // marcamos:
            filas[i] = true;
            d_asc[diag_asc(k, i)] = true;
            d_desc[diag_desc(k, i, n)] = true;
            // llamada recursiva:
            n_reinas(sol, k + 1, n, filas, d_asc, d_desc, cuantas_sol);
            // desmarcamos:
            filas[i] = false;
            d_asc[diag_asc(k, i)] = false;
            d_desc[diag_desc(k, i, n)] = false;
        }
    }
}

// función que resuelve el problema
int resolver(const int n) {
    int cuantas_sol = 0;
    vector<int> sol(n);
    int k = 0;
    // marcadores:
    vector<bool> filas(n);
    vector<bool> d_asc(2 * n - 1);
    vector<bool> d_desc(2 * n - 1);
    // llamada a la función:
    n_reinas(sol, k, n, filas, d_asc, d_desc, cuantas_sol);

    return cuantas_sol;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {
    // leer los datos de la entrada
    int n; cin >> n;

    int sol = resolver(n);

    // escribir sol
    cout << sol << '\n';
}

```

Compra de la semana

Enunciado:

Alonso Rodríguez tiene que hacer la compra de la semana. Ha hecho una lista de n productos que quiere comprar. En su barrio hay m supermercados en cada uno de los cuales se dispone de todos esos productos. Pero como es un comprador compulsivo no quiere comprar más de tres productos en cada uno de los supermercados ya que así pasa más tiempo comprando (se puede suponer que $n \leq 3m$). Dispone de una lista de precios (en céntimos) de los productos en cada uno de los supermercados. Se pide diseñar un algoritmo que permita a Alonso decidir cómo hacer la compra de forma que compre todo lo que necesita y que el coste total sea mínimo.

Entrada

La entrada comienza por una línea indicando el número de casos de prueba que deberán procesarse. Para cada caso de prueba la primera línea tiene dos números, el primero es el número de supermercados y el segundo el número de productos. Se garantiza que $0 \leq$ número de productos $\leq 3 *$ número de supermercados y que número de supermercados ≤ 20 . A continuación aparecen tantas líneas como supermercados y en cada una de las líneas el precio de todos los productos en ese supermercado. En todos los supermercados se ofrecen todos los productos.

Salida

Por cada caso de prueba aparecerá una línea independiente con el coste de la mejor solución encontrada o bien el mensaje "Sin solución factible" en el caso de que no haya ninguna.

Entrada de ejemplo

```
2
6 10
1820 510 370 1000 460 324 505 640 2030 409
2000 430 450 1110 606 290 530 670 2104 501
1760 502 395 1200 550 199 525 702 1830 550
2130 640 560 1307 735 450 600 720 2150 575
1143 455 505 1140 500 400 350 550 2030 399
1200 475 403 1002 560 350 502 640 2009 460
4 1
4020
3560
5540
3540
```

Salida de ejemplo

```
6743
3540
```

Solución SIN PODA:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ACLARACIONES
 *   . n_prod = nº de productos
 *   . n_super = nº de supermercados
 *
 *   ESPACIO DE BÚSQUEDA
 *
 *   El espacio de búsqueda es un árbol en el que la solución es una tupla (X0,...,Xn-1), donde n = n_prod
 *   y ParaTodo k : 0 ≤ k < n : (Xk = sol[k] = supermercado en el cual compramos el producto k) ∧ 0 ≤ Xk < n_super
 *
 *   MARCADORES
 *
 *   Los marcadores empleados para mejorar la eficiencia de las funciones es_solución() y es_completable() son los siguientes:
 *   . marcador: ParaTodo i : 0 ≤ i < n_super : marcador[i] = cuántos productos hemos comprado en el supermercado i
 */
const int MAX_PROD_POR_SUPER = 3;

// añade el coste del producto k en el supermercado sol[k] al coste que llevábamos hasta ahora
int actualiza(const int coste, vector<vector<int>> const& precios, const int i, const int k) {
    return coste + precios[i][k];
}

// vuelve al coste anterior
int desactualiza(const int coste, vector<vector<int>> const& precios, const int i, const int k) {
    return coste - precios[i][k];
}

/* comprueba que hemos decidido el supermercado para todos los productos y que en el último supermercado
   decidido no hemos comprado más de 3 productos (los otros supermercados se suponen ya comprobados) */
bool es_solución(const int n_prod, const int k, vector<int> const& sol, vector<int> const& marcador) {
    return k == n_prod - 1 && marcador[sol[k]] < MAX_PROD_POR_SUPER;
}
```

```

/* la nueva solución es mejor si no teníamos ninguna hasta ahora o si el coste calculado es menor que el
   que tenemos hasta ahora */
bool mejor(const int coste, const int mejor_coste) {
    return mejor_coste == -1 || coste < mejor_coste;
}

/* comprueba que todavía quedan productos de los que hay que determinar su supermercado y que en el último supermercado
   decidido no hemos comprado más de 3 productos (los otros supermercados se suponen ya comprobados) */
bool es_completable(const int n_prod, const int k, vector<int> const& sol, vector<int> const& marcador) {
    return k < n_prod - 1 && marcador[sol[k]] < MAX_PROD POR SUPER;
}

// algoritmo vuelta-atrás
void coste_minimo(vector<vector<int>> const& precios, const int n_super, const int n_prod, const int k, vector<int>& sol, int& coste, vector<int>& mejor_sol, int& mejor_coste, vector<int>& marcador) {
    for (int i = 0; i < n_super; ++i) {
        sol[k] = i; // el producto k lo compramos en el supermercado i
        coste = actualiza(coste, precios, i, k);
        if (es_solución(n_prod, k, sol, marcador) && mejor(coste, mejor_coste)) {
            mejor_sol = sol;
            mejor_coste = coste;
        }
        else if (es_completable(n_prod, k, sol, marcador)) {
            // marcamos:
            //+marcador[i];
            // vuelta atrás:
            coste_minimo(precios, n_super, n_prod, k + 1, sol, coste, mejor_sol, mejor_coste, marcador);
            // desmarcamos:
            //~marcador[i];
        }
        coste = desactualiza(coste, precios, i, k);
    }
}

// función que resuelve el problema
int resolver(vector<vector<int>> const& precios, const int n_super, const int n_prod) {
    int coste = 0, mejor_coste = -1;
    int k = 0;
    vector<int> sol(n_prod), mejor_sol(n_prod);
    vector<int> marcador(n_super, 0);
    coste_minimo(precios, n_super, n_prod, k, sol, coste, mejor_sol, mejor_coste, marcador);

    return mejor_coste;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {
    // leer los datos de la entrada
    int n_super, n_prod;
    cin >> n_super >> n_prod;
    vector<vector<int>> precios(n_super, vector<int>(n_prod));
    for (int i = 0; i < n_super; ++i)
        for (int j = 0; j < n_prod; ++j)
            cin >> precios[i][j];

    int sol = resolver(precios, n_super, n_prod);

    // escribir sol
    if (sol == -1) cout << "Sin solución factible";
    else cout << sol;
    cout << '\n';
}

```

Solución CON PODA:

```

// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *  ACLARACIONES
 *  - n_prod = nº de productos
 *  - n_super = nº de supermercados
 *
 *  ESPACIO DE BÚSQUEDA
 *  -----
 *  El espacio de búsqueda es un árbol en el que la solución es una tupla (X0,...,Xn-1), donde n = n_prod
 *  y ParaTodo k : 0 <= k < n : (X_k = sol[k] = supermercado en el cual compramos el producto k) /\ 0 <= X_k < n_super
 */

```

```

/*
 * MARCADORES
 *
 * Los marcadores empleados para mejorar la eficiencia de las funciones es_solucion() y es_completable() son los siguientes:
 * - marcador: ParaTodo i : 0 <= i < n_super : marcador[i] = cuántos productos hemos comprado en el supermercado i
 *
 * PODA
 *
 * Para mejorar el coste asintótico de la función se utiliza la siguiente función de poda:
 * Se emplea un vector precios_min donde se tiene almacenado el precio más barato que puede tener cada producto (el del supermercado
 * donde sea más barato). Así, cuando tenemos una solución parcial, si le añadimos el precio mínimo que pueden tener los productos que nos
 * quedan por comprar, obtenemos la mejor solución que podemos obtener a partir de la solución que llevamos. Si este valor es ya mayor que
 * el mejor que teníamos hasta ahora podamos el árbol, ya que no tiene sentido seguir mirando una solución que ni en el mejor de los casos
 * va a ser mejor que la que ya teníamos.
 *
 */

const int MAX_PROD_POR_SUPER = 3;

// añade el coste del producto k en el supermercado sol[k] al coste que llevábamos hasta ahora
int actualiza(const int coste, vector<vector<int>> const& precios, const int i, const int k) {
    return coste + precios[i][k];
}

// vuelve al coste anterior
int desactualiza(const int coste, vector<vector<int>> const& precios, const int i, const int k) {
    return coste - precios[i][k];
}

/* comprueba que hemos decidido el supermercado para todos los productos y que en el último supermercado
 * decidido no hemos comprado más de 3 productos (los otros supermercados se suponen ya comprobados) */
bool es_solucion(const int n_prod, const int k, vector<int> const& sol, vector<int> const& marcador) {
    return k == n_prod - 1 && marcador[sol[k]] < MAX_PROD_POR_SUPER;
}

/* la nueva solución es mejor si no teníamos ninguna hasta ahora o si el coste calculado es menor que el
 * que teníamos hasta ahora */
bool mejor(const int coste, const int mejor_coste) {
    return mejor_coste == -1 || coste < mejor_coste;
}

/* comprueba que todavía quedan productos de los que hay que determinar su supermercado y que en el último supermercado
 * decidido no hemos comprado más de 3 productos (los otros supermercados se suponen ya comprobados) */
bool es_completable(const int n_prod, const int k, vector<int> const& sol, vector<int> const& marcador) {
    return k < n_prod - 1 && marcador[sol[k]] < MAX_PROD_POR_SUPER;
}

/* comprueba si la mejor solución que podemos conseguir con lo que llevamos hasta ahora puede llegar a ser mejor
 * que la que teníamos hasta el momento (si no llevamos ninguna siempre es prometedora) */
bool es_prometedora(const int coste, const int mejor_coste, const int min_coste) {
    // si compramos los productos que nos quedan en los supermercados más baratos
    int min_coste_restante = coste + min_coste;
    return mejor_coste == -1 || min_coste_restante < mejor_coste;
}

// algoritmo vuelta-atrás
void coste_minimo(vector<vector<int>> const& precios, const int n_super, const int n_prod, vector<int> const& precios_min, int min_coste, int k,
    vector<int>& sol, int& coste, vector<int>& mejor_sol, int& mejor_coste, vector<int>& marcador) {
    for (int i = 0; i < n_super; ++i) {
        sol[k] = i; // el producto k lo compramos en el supermercado i
        coste = actualiza(coste, precios, i, k);
        min_coste -= precios_min[k];
        if (es_solucion(n_prod, k, sol, marcador) && mejor(coste, mejor_coste)) {
            mejor_sol = sol;
            mejor_coste = coste;
        }
        else if (es_completable(n_prod, k, sol, marcador) && es_prometedora(coste, mejor_coste, min_coste)) {
            // marcamos:
            ++marcador[i];
            // vuelta atrás:
            coste_minimo(precios, n_super, n_prod, precios_min, min_coste, k + 1, sol, coste, mejor_sol, mejor_coste, marcador);
            // desmarcamos:
            --marcador[i];
        }
        coste = desactualiza(coste, precios, i, k);
        min_coste += precios_min[k];
    }
}

```

```

// función que resuelve el problema
int resolver(vector<vector<int>> const& precios, const int n_super, const int n_prod, vector<int> const& precios_min, int min_coste) {
    int coste = 0, mejor_coste = -1;
    int k = 0;
    vector<int> sol(n_prod), mejor_sol(n_prod);
    vector<int> marcador(n_super, 0);
    coste_minimo(precios, n_super, n_prod, precios_min, min_coste, k, sol, coste, mejor_sol, mejor_coste, marcador);

    return mejor_coste;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {
    // leer los datos de la entrada
    int n_super, n_prod;
    cin >> n_super >> n_prod;
    vector<vector<int>> precios(n_super, vector<int>(n_prod));
    vector<int> precios_min(n_prod, -1);
    int min_coste = 0;
    for (int i = 0; i < n_super; ++i)
        for (int j = 0; j < n_prod; ++j) {
            cin >> precios[i][j];
            if (precios_min[j] == -1 || precios[i][j] < precios_min[j]) {
                if (precios[i][j] < precios_min[j])
                    min_coste -= precios_min[j];
                min_coste += precios[i][j];
                precios_min[j] = precios[i][j];
            }
        }

    int sol = resolver(precios, n_super, n_prod, precios_min, min_coste);

    // escribir sol
    if (sol == -1) cout << "Sin solución factible";
    else cout << sol;
    cout << '\n';
}

```

Los funcionarios del Ministerio

Enunciado:

El Ministro de Desinformación y Decencia se ha propuesto hacer trabajar en firme a sus funcionarios, para lo que se ha sacado de la manga una serie de trabajos (tantos como funcionarios). A pesar de su ineficacia, todos los funcionarios son capaces de hacer cualquier trabajo, aunque unos tardan más que otros. En el Ministerio todos se conocen bien, por lo que se sabe cuánto tardará cada funcionario en realizar cada uno de los trabajos. Para justificar su puesto, Su Excelencia el Sr. Ministro desea conocer la asignación óptima de trabajos a funcionarios de modo que la suma total de tiempos sea *mínima*.



Entrada

La entrada consta de una serie de casos de prueba. Cada uno comienza con una línea con el número N de funcionarios y trabajos ($1 \leq N \leq 20$). A continuación aparecerán N líneas, una por funcionario, con N números (entre 1 y 10.000) que indican lo que tarda ese funcionario en realizar cada uno de los N trabajos.

La entrada terminará con un caso sin funcionarios, que no debe procesarse.

Salida

Para cada caso de prueba se escribirá una línea con la suma total de tiempos que tardarán los funcionarios en realizar los trabajos asignados según la asignación óptima.

Entrada de ejemplo

```
3
10 20 30
40 20 10
60 10 20
3
10 15 20
30 40 50
60 80 99
0
```

Salida de ejemplo

```
30
120
```

Autor: Alberto Verdejo.

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ACLARACIONES
 *   - n_trab = nº de trabajos = nº_func = nº de funcionarios = n
 *
 *   ESPACIO DE BÚSQUEDA
 *
 *   El espacio de búsqueda es un árbol en el que la solución es una tupla (X0,...,Xn-1)
 *   y ParaTodo k : 0 <= k < n : (X_k = sol[k] = funcionario al que se le asigna el trabajo k) /\ 0 <= X_k < n
 *
 *   MARCADORES
 *
 *   Los marcadores empleados para mejorar la eficiencia de las funciones es_solucion() y es_completable() son los siguientes:
 *   - marcador: ParaTodo i : 0 <= i < n : marcador[i] = el funcionario i tiene ya asignado un trabajo
 *
 *   PODA
 *
 *   Para mejorar el coste asintótico de la función se utiliza la siguiente función de poda:
 *   Se emplea un vector tiempos_min donde se almacena el tiempo mínimo en el que puede ser realizado cada trabajo (el del funcionario
 *   que lo haga más rápido). Así, cuando tenemos una solución parcial, si le añadimos el tiempo mínimo que pueden tardar el resto
 *   de trabajos que quedan, obtenemos la mejor solución que podemos obtener a partir de la solución que llevamos. Si este valor es ya
 *   mayor que lo que teníamos hasta ahora podamos el árbol, ya que no tiene sentido seguir mirando una solución que ni en el mejor de los casos
 *   va a ser mejor que la que ya teníamos.
 */

// añade el tiempo que tarda el funcionario i en hacer el trabajo k al tiempo total que llevábamos hasta ahora
int actualiza(const int tiempo, vector<vector<int>> const& horas_funcionarios, const int i, const int k) {
    return tiempo + horas_funcionarios[i][k];
}

// vuelve al tiempo anterior
int desactualiza(const int tiempo, vector<vector<int>> const& horas_funcionarios, const int i, const int k) {
    return tiempo - horas_funcionarios[i][k];
}

/* comprueba que todos los trabajos tienen asignados un funcionario, y que el último funcionario al que le hemos asignado el trabajo k
 * no tenía ya otro trabajo asignado (el resto de funcionarios se suponen ya comprobados) */
bool es_solucion(const int k, const int n, const int i, vector<bool> const& marcador) {
    return k == n - 1 && !marcador[i];
}
```

```

/* la nueva solución es mejor si no teníamos ninguna hasta ahora o si el tiempo calculado es menor que el
   que teníamos hasta ahora */
bool mejor(const int tiempo, const int mejor_tiempo) {
    return mejor_tiempo == -1 || tiempo < mejor_tiempo;
}

/* comprueba que todavía quedan trabajos sin asignar, y que el último funcionario al que le hemos asignado el trabajo k
   no tenía ya otro trabajo asignado (el resto de funcionarios se suponen ya comprobados) */
bool es_completable(const int k, const int n, const int i, vector<bool> const& marcador) {
    return k < n - 1 && !marcador[i];
}

/* comprueba si la mejor solución que podemos conseguir con lo que llevamos hasta ahora puede llegar a ser mejor
   que la que teníamos hasta el momento (si no llevamos ninguna siempre es prometedora) */
bool es_prometedora(const int tiempo, const int mejor_tiempo, const int min_tiempo) {
    // si los trabajos restantes los realizasen los funcionarios más rápidos en esos trabajos
    int min_tiempo_restante = tiempo + min_tiempo;
    return mejor_tiempo == -1 || min_tiempo_restante < mejor_tiempo;
}

// algoritmo vuelta-atrás
void tiempo_minimo(vector<vector<int>> const& horas_funcionarios, const int n, vector<int> const& tiempos_min, int min_tiempo, const int k,
                   vector<int>& sol, int& tiempo, vector<int>& mejor_sol, int& mejor_tiempo, vector<bool>& marcador) {

    for (int i = 0; i < n; ++i) {
        sol[k] = i; // el trabajo k lo realiza el funcionario i
        tiempo = actualiza(tiempo, horas_funcionarios, i, k);
        min_tiempo -= tiempos_min[k];
        if (es_solución(k, n, i, marcador) && mejor(tiempo, mejor_tiempo)) {
            mejor_sol = sol;
            mejor_tiempo = tiempo;
        }
        else if (es_completable(k, n, i, marcador) && es_prometedora(tiempo, mejor_tiempo, min_tiempo)) {
            // marcamos:
            marcador[i] = true;
            // vuelta atrás:
            tiempo_minimo(horas_funcionarios, n, tiempos_min, min_tiempo, k + 1, sol, tiempo, mejor_sol, mejor_tiempo, marcador);
            // desmarcamos:
            marcador[i] = false;
        }
        tiempo = desactualiza(tiempo, horas_funcionarios, i, k);
        min_tiempo += tiempos_min[k];
    }
}

// función que resuelve el problema
int resolver(vector<vector<int>> const& horas_funcionarios, vector<int> const& tiempos_min, int min_tiempo) {

    int k = 0;
    int n = horas_funcionarios.size();
    int tiempo = 0, mejor_tiempo = -1;
    vector<int> sol(n), mejor_sol(n);
    vector<bool> marcador(n, false);

    tiempo_minimo(horas_funcionarios, n, tiempos_min, min_tiempo, k, sol, tiempo, mejor_sol, mejor_tiempo, marcador);
    return mejor_tiempo;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    if (n == 0)
        return false;
    vector<vector<int>> horas_funcionarios(n, vector<int>(n));
    vector<int> tiempos_min(n, -1);
    int min_tiempo = 0;
    for (int i = 0; i < n; ++i)
        for (int j = 0; j < n; ++j) {
            cin >> horas_funcionarios[i][j];
}

```

```

    if (tiempos_min[j] == -1 || horas_funcionarios[i][j] < tiempos_min[j]) {
        if (horas_funcionarios[i][j] < tiempos_min[j])
            min_tiempo -= tiempos_min[j];
        min_tiempo += horas_funcionarios[i][j];
        tiempos_min[j] = horas_funcionarios[i][j];
    }

    int sol = resolver(horas_funcionarios, tiempos_min, min_tiempo);

    // escribir sol
    cout << sol << '\n';
    return true;
}

```

Noticias de Inglaterra (torres y reinas)

Enunciado:

En 1955 las noticias no vuelan, sino que tardan en llegar. Y en Hill Valley aún más. Ha habido que esperar más de un año para que por fin se hayan enterado de que el gran Alan Turing murió en junio del año pasado.

Aprovechando la noticia, en el Café de Lou se ha originado un debate sobre sus logros. En la conversación no ha salido, claro, nada sobre su trabajo rompiendo el cifrado de la máquina Enigma durante la segunda guerra mundial, pues esa es aún información clasificada que no se conocerá hasta los años setenta.

De lo que sí se ha hablado es de sus aportaciones a la inteligencia artificial (bueno, tampoco lo han llamado así, que el término se acuñó en 1956). En concreto, del programa que escribió para un ordenador que aún no existía y que hoy es considerado el primer programa para jugar al ajedrez por computador que se creó. La historia cuenta que dos años antes de morir Alan intentó implementarlo en el Mark 1, pero el ordenador no era capaz de ejecutar el programa. Así que para probar su funcionamiento, jugó una partida con su colega Alick Glennie simulando la ejecución con lápiz y papel.

Has visto tan interesada a la gente en el Café que te has puesto a explicarles que el programa de Alan Turing utilizaba un esquema parecido a la vuelta atrás inspeccionando una mínima parte del árbol de juego. Como te han mirado con cara de no haberse enterado de nada, has echado un poco el freno, vuelto atrás en las explicaciones y empezado por el ejemplo clásico del problema de las 8 reinas.

Y en un momento de entusiasmo has cambiado el problema y lo has hecho un poco más complicado... El enunciado que te ha salido es: ¿de cuántas formas se pueden colocar T torres y R reinas en un tablero de ajedrez sin que se ataquen? Además, algunas de las casillas del tablero están rotas y no se puede colocar en ellas ficha alguna. Y ahora te toca hacerlo y explicarlo a la audiencia...

Entrada

La entrada está formada por distintos casos de prueba. Para cada caso, la primera línea contiene dos naturales (T y R) indicando el número de torres y el número de reinas que hay que colocar en un tablero de $T + R$ filas y columnas (como mucho de tamaño 9×9). Tras esto aparece una línea indicando cuántas casillas del tablero no pueden utilizarse. Por último, aparece una línea por cada casilla inválida con su posición en formato fila-columna (números entre 1 y $T + R$).

Salida

Para cada caso de prueba se escribirá en una línea el número de formas posibles de colocar T torres y R reinas sin que se ataquen entre ellas de forma que, además, ninguna esté colocada en una casilla inválida.

Entrada de ejemplo

```
2 0
0
2 0
1
1 1
8 0
0
0 8
0
```

Salida de ejemplo

```
2
1
40320
92
```

Autor: Marco Antonio Gómez Martín.

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ACLARACIONES
 *   - T = nº de torres
 *   - R = nº de reinas
 *
 *   ESPACIO DE BÚSQUEDA
 *   -----
 *   El espacio de búsqueda es un árbol en el que la solución es una tupla  $(x_0, \dots, x_{n-1})$ , donde  $n = T + R$ 
 *   y ParaTodo  $k : 0 \leq k < n : (x_k = sol[k]) =$  fila en la que se coloca la reina/torre  $k$   $\wedge 0 \leq x_k < n$ 
 *
 *   MARCADORES
 *   -----
 *   - filas: ParaTodo  $i : 0 \leq i < n : filas[i] =$  la fila  $i$  está ocupada
 *   - diag_torres: ParaTodo  $i : 0 \leq i < 4n - 1 : diag_torres[i] =$  la diagonal  $i$  está ocupada por una torre
 *   - diag_reinas: ParaTodo  $i : 0 \leq i < 4n - 1 : diag_reinas[i] =$  la diagonal  $i$  está ocupada por una reina
 */

using matriz = vector<vector<bool>>;
using marcador = vector<bool>;

// calcula la diagonal ascendente de la casilla [f,c]
int d_asc(const int f, const int c, const int n) {
    return c + f + 2 * n - 1;
}

// calcula la diagonal descendente de la casilla [f,c]
int d_desc(const int f, const int c, const int n) {
    return c - f + n - 1;
}
```

```

// comprueba que la casilla no está rota, que no había ya una figura en esa fila, y que no había una reina ya en esa diagonal
bool casilla_valida(const int& invalidas, const int fila, const int col, const int n, const int& filas, const int& diag_reinas) {
    return !invalidas[fila][col] && !filas[fila] && !diag_reinas[d_asc(fila, col, n)] && !diag_reinas[d_desc(fila, col, n)];
}

// comprueba que hemos colocado todas las figuras
bool es_solucion(const int k, const int n) {
    return k == n - 1;
}

// comprueba que todavía quedan figuras por colocar
bool es_completable(const int k, const int n) {
    return k < n - 1;
}

// comprueba si ya había torres en las diagonales de la casilla
bool hay_torres_diag(const int k, const int n, vector<int>& sol, const int& filas, const int& diag_torres) {
    return diag_torres[d_asc(sol[k], k, n)] || diag_torres[d_desc(sol[k], k, n)];
}

// marca o desmarca todos los marcadores pasados como entrada
void marcar_desmarcar(const bool marca_f, const bool marca_d_asc, const bool marca_d_desc, const int& filas, const int& diagonales, const int fila, const int col, const int n) {
    filas[fila] = marca_f;
    diagonales[d_asc(fila, col, n)] = marca_d_asc;
    diagonales[d_desc(fila, col, n)] = marca_d_desc;
}

// algoritmo vuelta atrás
void torres_reinas(int& torres, int& reinas, const int n, matriz const& invalidas, int k, vector<int>& sol, int& cuantas_sol, const int& filas, const int& diagonales, const int& diag_torres, const int& diag_reinas) {
    for (int i = 0; i < n; ++i) {
        if (casilla_valida(invalidas, i, k, n, filas, diag_reinas)) {
            sol[k] = i; // colocamos figura en la pos [i, k]
            // colocamos torre:
            if (torres > 0) {
                --torres;
                if (es_solucion(k, n))
                    ++cuantas_sol;
                else if (es_completable(k, n)) {
                    // guardamos marcadores anteriores por si ya había una torre en esa diagonal
                    bool anterior_asc = diag_torres[d_asc(i, k, n)], anterior_desc = diag_torres[d_desc(i, k, n)];
                    marcar_desmarcar(true, true, true, filas, diag_torres, i, k, n);
                    torres_reinas(torres, reinas, n, invalidas, k + 1, sol, cuantas_sol, filas, diag_torres, diag_reinas);
                    marcar_desmarcar(false, anterior_asc, anterior_desc, filas, diag_torres, i, k, n);
                }
                ++torres;
            }
            // colocamos reina:
            if (reinas > 0) {
                --reinas;
                if (es_solucion(k, n) && !hay_torres_diag(k, n, sol, diag_torres))
                    ++cuantas_sol;
                else if (es_completable(k, n) && !hay_torres_diag(k, n, sol, diag_torres)) {
                    marcar_desmarcar(true, true, true, filas, diag_reinas, i, k, n);
                    torres_reinas(torres, reinas, n, invalidas, k + 1, sol, cuantas_sol, filas, diag_torres, diag_reinas);
                    marcar_desmarcar(false, false, false, filas, diag_reinas, i, k, n);
                }
                ++reinas;
            }
        }
    }
}

// función que resuelve el problema
int resolver(const int T, const int R, const int N, matriz const& invalidas) {
    int torres = T, reinas = R;
    int k = 0;
    vector<int> sol(N);
    int cuantas_sol = 0;
    if (N > 0) {
        marcador filas(N, false);
        marcador diag_torres(4 * N - 2, false);
        marcador diag_reinas(4 * N - 2, false);

        torres_reinas(torres, reinas, N, invalidas, k, sol, cuantas_sol, filas, diag_torres, diag_reinas);
    }

    return cuantas_sol;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int T; cin >> T;
    if (!cin)
        return false;
    int R; cin >> R;
    int N = T + R; // tamaño tablero
    int I; cin >> I;
    matriz invalidas(N, vector<bool>(N, false));
    int fila, col;
}

```

```

    for (int i = 0; i < I; ++i) {
        cin >> fila;
        cin >> col;
        invalidas[fila - 1][col - 1] = true;
    }

    int sol = resolver(T, R, N, invalidas);

    // escribir sol
    cout << sol << '\n';

    return true;
}

```

Papa Noel reparte juguetes (simplificado)

Enunciado:

Cada año son más los niños que le piden regalos a Papa Noel. Esta noche debe repartir los regalos y todavía no tiene preparado lo que le dará a cada niño. Para poder llegar a tiempo los elfos han diseñado un programa informático, que asignará a cada niño 2 juguetes. Se tienen todas las unidades que hagan falta de cada juguete. Papa Noel ha pedido que los juguetes sean variados, para ello los elfos han clasificado los juguetes por tipos y no entregarán a un niño dos juguetes del mismo tipo. Por ejemplo, los juguetes *camión de bomberos* y *coche de policía* pertenecen al mismo tipo *Coche*, así que no podemos darle al mismo niño los dos juguetes, aunque tenemos infinitas unidades de camiones de bomberos y de coches de policías.

El jefe elfo de informática ha puesto a su equipo a trabajar en un programa que obtenga todas las posibles asignaciones de juguetes que cumplen los requisitos pedidos por Papa Noel. Para evitar las asignaciones repetidas (por ejemplo dar a un niño el juguete A y el juguete B es la misma asignación que darle el juguete B y el juguete A) los dos juguetes de cada niño se obtendrán ordenados, el primer juguete será el de menor identificador y el segundo el de mayor identificador.



Requisitos de implementación.

La solución se almacena en un vector donde las componentes pares tienen el primer juguete asignado a un niño y las componentes impares el segundo juguete. En las etapas pares se asigna el primer juguete de un niño y en las etapas impares el segundo juguete.

Para evitar las asignaciones repetidas, se comprueba en la función `esValida` que el identificador del primer juguete sea menor que el identificador del segundo.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de 2 líneas. En la primera se indica el número juguetes que se fabrican, *m*, y el número de niños a los que se les reparten juguetes, *n*. En la siguiente línea se indica el tipo que tiene cada juguete.

El número de juguetes, y el número de niños son enteros mayores que cero. El tipo de cada juguete es una cadena de caracteres.

Salida

Para cada caso de prueba se escriben todas las posibles asignaciones que se pueden realizar, una en cada línea. Si no existe ninguna asignación posible se escribirá SIN SOLUCION. Después de cada caso se escribe una línea en blanco.

Entrada de ejemplo

```
3 1
Tipo1 Tipo2 Tipo3
3 1
Tipo1 Tipo1 Tipo2
3 1
Tipo1 Tipo1 Tipo1
2 2
Jug1 Jug2
3 2
Coche Coche Libro
```

Salida de ejemplo

```
0 1
0 2
1 2

0 2
1 2

SIN SOLUCION

0 1 0 1

0 2 0 2
0 2 1 2
1 2 0 2
1 2 1 2
```

Autor: Isabel Pita

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ESPACIO DE BÚSQUEDA
 *
 *   El espacio de búsqueda es un árbol en el que la solución es una tupla (X0,...,Xn-1), donde n = 2 * num_ninyos
 *   y ParaTodo k : 0 <= k < n : (k % 2 == 0 --> X_k = sol[k] = primer regalo que recibe el niño k / 2) /\ (k % 2 == 1 --> X_k = sol[k] = segundo regalo que recibe el niño k / 2) /\ 
 *   /\ 0 <= X_k < num_juguetes
 *
 *   MARCADORES
 *
 *   - marcador: ParaTodo i : 0 <= i < num_ninyos : marcador[i] = tipo del primer regalo que ha recibido el niño i (" " si no ha recibido todavía ninguno)
 *
 */

/* comprueba que, o bien estamos asignando el primer regalo, o bien estamos asignando el segundo y no hemos hecho repeticiones ni estamos dando un regalo del mismo tipo que ya tenía el último niño (el resto se suponen comprobados) */
bool es_valida(const int k, const int i, const vector<int> &sol, const vector<string> &tipos, const vector<string> &marcador) {
    return k % 2 == 0 || (sol[k - 1] < i && tipos[i] != marcador[k / 2]);
}

// comprueba que todos los niños tienen ya sus 2 juguetes
bool es_solucion(const int k, const int n) {
    return k == n - 1;
}

// comprueba que quedan niños sin todos sus juguetes
bool es_completable(const int k, const int n) {
    return k < n - 1;
}

// imprime la solución
void procesar_solucion(const vector<int> &sol, const int n) {
    for (int i = 0; i < n; ++i)
        cout << sol[i] << " ";
    cout << "\n";
}
```

```

// algoritmo vuelta atrás
void papaNoel(bool& hay_solucion, vector<string> const& tipos, const int num_juguetes, const int k, const int n, vector<int>& sol, vector<string>& marcador) {
    for (int i = 0; i < num_juguetes; ++i) {
        if (es_valida(k, i, sol, tipos, marcador)) {
            sol[k] = i;
            if (es_solucion(k, n)) {
                hay_solucion = true;
                procesar_solucion(sol, n);
            }
            else if (es_completable(k, n)) {
                marcador[k / 2] = tipos[i];
                papaNoel(hay_solucion, tipos, num_juguetes, k + 1, n, sol, marcador);
                marcador[k / 2] = "";
            }
        }
    }
}

// función que resuelve el problema
bool resolver(const int num_juguetes, const int num_ninyos, vector<string> const& tipos) {
    bool hay_solucion = false;
    int k = 0, n = num_ninyos * 2;
    vector<int> sol(n);
    vector<string> marcador(num_ninyos, "");

    papaNoel(hay_solucion, tipos, num_juguetes, k, n, sol, marcador);

    return hay_solucion;
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int num_juguetes; cin >> num_juguetes;
    if (!std::cin)
        return false;
    int num_ninyos; cin >> num_ninyos;
    vector<string> tipos(num_juguetes);
    for (int i = 0; i < num_juguetes; ++i)
        cin >> tipos[i];

    bool sol = resolver(num_juguetes, num_ninyos, tipos);

    // escribir sol
    if (!sol) cout << "SIN SOLUCION" << '\n';
    cout << '\n';
    return true;
}

```

Papa Noel reparte juguetes (optimización)

Enunciado:

Cada año son más los niños que le piden regalos a Papa Noel. Esta noche debe repartir los regalos y todavía no tiene preparado lo que le dará a cada niño. Para poder llegar a tiempo los elfos han diseñado un programa informático, que asignará a cada niño un juguete entre todos los disponibles. Cada juguete sólo se puede asignar a un niño. Como quieren que los niños queden contentos han elaborado una lista con la satisfacción que le produce a cada niño cada uno de los juguetes que tienen en la fábrica. El objetivo es maximizar el grado de satisfacción del conjunto de todos los niños. La suma de la satisfacción de todos los niños debe ser máxima.

El jefe elfo de informática ha puesto a su equipo a trabajar en un programa que obtenga la satisfacción máxima que pueden conseguir.

Entrada



La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de $n+1$ líneas. En la primera se indica el número juguetes que se fabrican, m ($1 \leq m \leq 10$), y el número de niños a los que se les reparten juguetes, n ($1 \leq n \leq 9$). En las n líneas siguientes se indica la satisfacción de cada uno de los n niños con cada uno de los m juguetes. Después de cada caso hay una línea en blanco para facilitar la identificación de los casos en el ejemplo.

Se garantiza que el número de juguetes es siempre igual o superior al número de niños. La satisfacción es un número entero que puede ser negativo si el niño aborrece el juguete.

Salida

Para cada caso de prueba se escribe la satisfacción máxima que se puede conseguir.

Entrada de ejemplo

```
4 3
8 9 3 1
6 4 5 3
2 2 9 9

4 3
8 9 3 1
6 4 5 3
2 2 9 9

4 2
10 12 12 15
9 10 20 7
```

Salida de ejemplo

```
24
24
35
```

Autor: Isabel Pita

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ESPACIO DE BÚSQUEDA
 *
 *   -----
 *   El espacio de búsqueda es un árbol en el que la solución es una tupla  $(X_0, \dots, X_{n-1})$ , donde  $n = \text{num_ninos}$ 
 *   y ParaTodo  $k : 0 \leq k < n : (X_k = \text{sol}[k] = \text{regalo que le damos al niño } k) \wedge 0 \leq X_k < \text{num_juguetes}$ 
 *
 *   MARCADORES
 *
 *   -----
 *   . marcador: ParaTodo  $i : 0 \leq i < \text{num_juguetes} : \text{marcador}[i] = \text{ya hemos asignado el juguete } i$ 
 *
 *   PODA
 *
 *   -----
 *   La función de poda consiste en llevar una variable que almacene la mejor satisfacción posible que podemos
 *   obtener con lo que llevamos, para poder podar si aún en el mejor de los casos no conseguiríamos superar a la mejor
 *   solución que llevamos hasta el momento. Para ello se utiliza un vector que lleva para cada regalo, cuál es la mayor
 *   satisfacción que puede obtenerse.
 *
 */

using matriz = vector<vector<int>>;
```

```

// comprueba que no habíamos repartido ya ese regalo
bool es_valida(const int i, vector<bool> const& marcador) {
    return !marcador[i];
}

// comprueba que hemos repartido todos los regalos
bool es_solucion(const int k, const int n) {
    return k == n - 1;
}

// comprueba que la solución encontrada es mejor que la que teníamos hasta ahora
bool es_mejor(const int sat_actual, const int sat_max) {
    return sat_actual > sat_max;
}

// comprueba que todavía quedan regalos por repartir
bool es_completarable(const int k, const int n) {
    return k < n - 1;
}

// comprueba si la mejor solución que podríamos llegar a obtener con lo que llevamos hasta ahora es mejor que la sat_max que teníamos
bool es_prometedora(const int sat_actual, const int max_sat_restante, const int sat_max) {
    return sat_actual + max_sat_restante > sat_max;
}

// algoritmo vuelta atrás
void papa_noel_sat_max(matriz const& satisfacciones, const int num_juguetes, const int num_ninyos, const int k, int& sat_actual,
int& sat_max, vector<bool>& marcador, vector<int> const& max_satisfacciones, int& max_sat_restante) {

    for (int i = 0; i < num_juguetes; ++i) {
        if (es_valida(i, marcador)) {
            sat_actual += satisfacciones[k][i];
            max_sat_restante -= max_satisfacciones[i];
            if (es_solucion(k, num_ninyos) && es_mejor(sat_actual, sat_max))
                sat_max = sat_actual;
            else if (es_completarable(k, num_ninyos) && es_prometedora(sat_actual, max_sat_restante, sat_max)) {
                marcador[i] = true;
                papa_noel_sat_max(satisfacciones, num_juguetes, num_ninyos, k + 1, sat_actual, sat_max, marcador, max_satisfacciones, max_sat_restante);
                marcador[i] = false;
            }
            sat_actual -= satisfacciones[k][i];
            max_sat_restante += max_satisfacciones[i];
        }
    }
}

// función que resuelve el problema
int resolver(const int num_juguetes, const int num_ninyos, matriz const& satisfacciones, vector<int> const& max_satisfacciones, int& max_sat_restante) {

    int k = 0;
    int sat_actual = 0, sat_max = 0;
    vector<bool> marcador(num_juguetes, false);

    papa_noel_sat_max(satisfacciones, num_juguetes, num_ninyos, k, sat_actual, sat_max, marcador, max_satisfacciones, max_sat_restante);

    return sat_max;
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int num_juguetes; cin >> num_juguetes;
    if (!std::cin)
        return false;
    int num_ninyos; cin >> num_ninyos;
    matriz satisfacciones(num_ninyos, vector<int>(num_juguetes));
    vector<int> max_satisfacciones(num_juguetes, -1);
    int max_sat_restante = 0;
    for (int i = 0; i < num_ninyos; ++i)
        for (int j = 0; j < num_juguetes; ++j) {
            cin >> satisfacciones[i][j];
            if (max_satisfacciones[j] == -1 || max_satisfacciones[j] < satisfacciones[i][j]) {
                if (max_satisfacciones[j] != -1)
                    max_sat_restante -= max_satisfacciones[j];
                max_satisfacciones[j] = satisfacciones[i][j];
                max_sat_restante += max_satisfacciones[j];
            }
        }
}

```

```

int sol = resolver(num_juguetes, num_ninyos, satisfacciones, max_satisfacciones, max_sat_restante);

// escribir sol
cout << sol << '\n';
return true;
}

```

Adornando la casa por Navidad

Enunciado:

Este año mis padres han decidido renovar los adornos que tradicionalmente ponemos en la casa por Navidad. Quieren comprar algunas figuras para el jardín delantero y para colgar en la fachada; un enorme árbol del que cuelguen bolas, luces y muñecos; y varias guirnaldas para las paredes y techos. Han seleccionado en un catálogo todos aquellos objetos que les gustaría poner. Ahora quieren maximizar la superficie cubierta por los adornos, sin sobrepasar el presupuesto que tienen. Debemos tener en cuenta que los adornos no se pueden partir, ya que correríamos el riesgo de provocar un cortocircuito eléctrico.



Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de $n + 1$ líneas. En la primera se indica el número de objetos que se pueden comprar ($1 \leq n \leq 30$) y el presupuesto con que se cuenta. En las n siguientes se indican el coste y la superficie que ocupa cada uno de los objetos.

Salida

Para cada caso de prueba se escribe en una línea la máxima superficie que pueden cubrir.

Entrada de ejemplo

```

2 10
1 1
10 4
4 10
4 2
3 4
5 5
2 1
4 10
4 6
3 4
5 5
3 1
8 15
7 6
5 4
10 8
8 9
6 8
5 6
7 5
6 8

```

Salida de ejemplo

```

4
10
11
17

```

Salida del ejemplo mostrando un vector solución. En la primera línea se muestra una asignación óptima y en la línea siguiente el coste de esta solución. Esta forma de escribir los datos NO es la utilizada en el juez, solo se da para ayudar a depurar el programa.

Salida de ejemplo

```
1
4

1 2 3
10

0 1 3
11

3 4
17

0 5 7
20
```

Autor: Isabel Pita.

Solución SIN PODA:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ESPACIO DE BÚSQUEDA
 *
 *   El espacio de búsqueda es un árbol binario en el que la solución es una tupla (X0,...,Xn-1), donde n = nº de adornos
 *   y ParaTodo k : 0 <= k < n : X_k = sol[k] = ponemos el adorno k
 *
 *   MARCADORES
 *
 *   - precio_actual = precio que llevamos acumulado hasta ahora con los adornos que hemos puesto
 *   - sup_actual = superficie que llevamos cubierta con los adornos que hemos puesto
 */
using objeto = pair<int, int>;

// comprueba que no nos hemos pasado del presupuesto
bool es_valida(const int precio_actual, const int presupuesto) {
    return precio_actual <= presupuesto;
}

// comprueba que hemos decidido sobre todos los adornos
bool es_solucion(const int k, const int n) {
    return k == n - 1;
}

// comprueba si la solución que hemos obtenido es mejor que la que llevábamos hasta ahora
bool es_mejor(const int sup_actual, const int max_sup) {
    return sup_actual > max_sup;
}

// comprueba que quedan adornos por decidir si poner o no
bool es_completable(const int k, const int n) {
    return k < n - 1;
```

```

// algoritmo vuelta atrás
void decorar(const int presupuesto, const int n, vector<objeto> const& objetos, const int k, int& precio_actual, int& sup_actual, int& max_sup) {
    // probamos a no cogerlo:
    if (es_solucion(k, n) && es_mejor(sup_actual, max_sup))
        max_sup = sup_actual;
    else if (es_completable(k, n))
        decorar(presupuesto, n, objetos, k + 1, precio_actual, sup_actual, max_sup);

    // probamos a cogerlo:
    precio_actual += objetos[k].first;
    sup_actual += objetos[k].second;
    if (es_valida(precio_actual, presupuesto)) {
        if (es_solucion(k, n) && es_mejor(sup_actual, max_sup))
            max_sup = sup_actual;
        else if (es_completable(k, n))
            decorar(presupuesto, n, objetos, k + 1, precio_actual, sup_actual, max_sup);
    }
    precio_actual -= objetos[k].first;
    sup_actual -= objetos[k].second;
}

// función que resuelve el problema
int resolver(const int n, vector<objeto> const& objetos, const int presupuesto) {
    int k = 0;
    int precio_actual = 0;
    int sup_actual = 0, max_sup = 0;

    decorar(presupuesto, n, objetos, k, precio_actual, sup_actual, max_sup);

    return max_sup;
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    if (!std::cin)
        return false;
    int presupuesto; cin >> presupuesto;
    vector<objeto> objetos(n);
    for (int i = 0; i < n; ++i)
        cin >> objetos[i].first >> objetos[i].second;

    int sol = resolver(n, objetos, presupuesto);

    // escribir sol
    cout << sol << '\n';
    return true;
}

```

Solución CON PODA:

```

// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
#include <algorithm>
using namespace std;

/*
 *   ESPACIO DE BÚSQUEDA
 * -----
 *   El espacio de búsqueda es un árbol binario en el que la solución es una tupla (X0,...,Xn-1), donde n = nº de adornos
 *   y ParaTodo k : 0 <= k < n : X_k = sol[k] = ponemos el adorno k
 *
 *   MARCADORES
 * -----
 *   - precio_actual = precio que llevamos acumulado hasta ahora con los adornos que hemos puesto
 *   - sup_actual = superficie que llevamos cubierta con los adornos que hemos puesto
 */

```

```

/*
 * Poda
 * -----
 * Se ordena en orden decreciente el vector de objetos según el cociente superficie/coste, y se estima la máxima superficie
 * restante que se puede llegar a conseguir para poder podar soluciones que no llegarán a ser mejores que la que tenemos. Además,
 * se mejora la eficiencia inicializando la mejor solución a una solución inicial (en vez de 0), aquella que coge todos los objetos
 * que puede sin pasarse del presupuesto.
 */

using objeto = pair<int, int>

// comprueba que no nos hemos pasado del presupuesto
bool es_valida(const int precio_actual, const int presupuesto) {
    return precio_actual <= presupuesto;
}

// comprueba que hemos decidido sobre todos los adornos
bool es_solucion(const int k, const int n) {
    return k == n - 1;
}

// comprueba si la solución que hemos obtenido es mejor que la que llevábamos hasta ahora
bool es_mejor(const int sup_actual, const int max_sup) {
    return sup_actual > max_sup;
}

// comprueba que quedan adornos por decidir si poner o no
bool es_completetable(const int k, const int n) {
    return k < n - 1;
}

// comprueba si con lo que llevamos hasta ahora podemos llegar a obtener una solución mejor a la que teníamos
bool es_prometedora(vector<objeto> const& objetos, const int n, const int k, const int precio_actual, const int presupuesto, const int sup_actual, const int max_sup) {
    int sup_restante = 0, coste = precio_actual;
    int cont = k + 1;

    while (cont < n && coste < presupuesto) {
        sup_restante += objetos[cont].second;
        coste += objetos[cont].first;
        ++cont;
    }

    // si nos hemos pasado cortamos el último objeto que hemos cogido
    if (cont < n + 1 && coste > presupuesto)
        sup_restante -= objetos[cont - 1].second * (coste - presupuesto) / objetos[cont - 1].first;

    return sup_actual + sup_restante > max_sup;
}

// algoritmo vuelta atrás
void decorar(const int presupuesto, const int n, vector<objeto> const& objetos, const int k, int& precio_actual, int& sup_actual, int& max_sup) {
    // probamos a no cogerlo:
    if (es_solucion(k, n) && es_mejor(sup_actual, max_sup))
        max_sup = sup_actual;
    else if (es_completetable(k, n) && es_prometedora(objetos, n, k, precio_actual, presupuesto, sup_actual, max_sup))
        decorar(presupuesto, n, objetos, k + 1, precio_actual, sup_actual, max_sup);

    // probamos a cogerlo:
    precio_actual += objetos[k].first;
    sup_actual += objetos[k].second;
    if (es_valida(precio_actual, presupuesto)) {
        if (es_solucion(k, n) && es_mejor(sup_actual, max_sup))
            max_sup = sup_actual;
        else
            decorar(presupuesto, n, objetos, k + 1, precio_actual, sup_actual, max_sup);
    }
    precio_actual -= objetos[k].first;
    sup_actual -= objetos[k].second;
}

// función que resuelve el problema
int resolver(const int n, vector<objeto> const& objetos, const int presupuesto, const int sup_inicial) {
    int k = 0;
    int precio_actual = 0;
    int sup_actual = 0, max_sup = sup_inicial;
    decorar(presupuesto, n, objetos, k, precio_actual, sup_actual, max_sup);
    return max_sup;
}

```

```

// calcula la solución inicial
int inicializar_sol(vector<objeto> const& objetos, const int n, const int presupuesto) {
    int sup_inicial = 0, coste = 0;
    int cont = 0;

    while (cont < n && coste < presupuesto) {
        sup_inicial += objetos[cont].second;
        coste += objetos[cont].first;
        ++cont;
    }

    // si nos hemos pasado quitamos el último objeto
    if (cont < n + 1 && coste > presupuesto)
        sup_inicial -= objetos[cont - 1].second;

    return sup_inicial;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {
    // leer los datos de la entrada
    int n; cin >> n;
    if (!std::cin)
        return false;
    int presupuesto; cin >> presupuesto;
    vector<objeto> objetos(n);
    for (int i = 0; i < n; ++i)
        cin >> objetos[i].first >> objetos[i].second;

    // ordenamos los objetos por índice superficie/coste
    sort(objetos.begin(), objetos.end(), [](const objeto o1, const objeto o2) { return (float)o1.second/o1.first > (float)o2.second/o2.first; });

    // calculamos sup_inicial
    int sup_inicial = inicializar_sol(objetos, n, presupuesto);

    int sol = resolver(n, objetos, presupuesto, sup_inicial);

    // escribir sol
    cout << sol << '\n';
    return true;
}

```

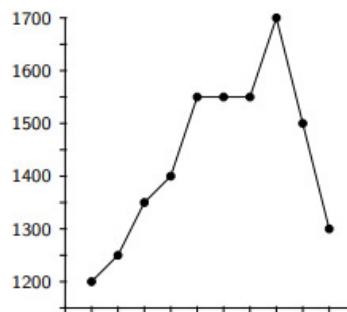
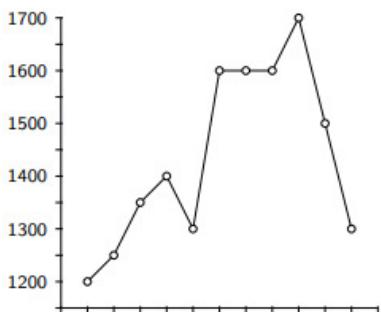
Caminando voy

Enunciado:

Con la llegada del buen tiempo, el grupo de senderistas *Caminando voy* quiere preparar una serie de excursiones por la Sierra de Madrid. Las excursiones deben ser aptas para todos los socios (hay un numeroso grupo de jubilados y otro de niños de corta edad).

En concreto, se requiere que las cuestas arriba no se hagan demasiado penosas. Para ello, para cada excursión se ha confeccionado un perfil de desniveles, consistente en una secuencia de cotas de altura (valores enteros no negativos). Las cuestas arriba se corresponden con segmentos estrictamente crecientes, y el desnivel de una cuesta será la diferencia entre su cota más alta y su cota más baja. Una excursión se considerará *apta* si en todas las cuestas arriba el desnivel no supera un cierto valor estipulado $D \geq 0$.

Por ejemplo, suponiendo $D = 300$, una excursión con el perfil de la izquierda sería apta, mientras que una con el perfil de la derecha no.



Ahora necesitan un programa que clasifique todas las excursiones que tienen programadas.

Entrada

La entrada consta de una serie de casos de prueba, cada uno correspondiente a una excursión. Cada caso ocupa dos líneas. En la primera aparecen dos números separados por un espacio: el desnivel máximo permitido D (entre 0 y 1.000.000) y el número N (entre 1 y 200.000) de cotas de altura que forman la excursión. En la segunda línea aparecen las N cotas (números entre 0 y 1.000.000), separadas por espacios.

Salida

Para cada caso de prueba, el programa escribirá APTA si la excursión es apta para el desnivel máximo permitido, y NO APTA en caso contrario.

Entrada de ejemplo

```
300 11
1200 1250 1350 1400 1300 1600 1600 1600 1700 1500 1300
300 10
1200 1250 1350 1400 1550 1550 1550 1700 1500 1300
0 4
10 10 10 10
```

Salida de ejemplo

```
APTA
NO APTA
APTA
```

Autores: Yolanda Ortega Mallén y Alberto Verdejo.

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 * ACLARACIONES
 * - N = cotas.size()
 *
 * ESPECIFICACIÓN
 * -----
 * P = { 0 <= D <= 1.000.000 & 1 <= N <= 200.000 & (ParaTodo w : 0 <= w < N : 0 <= cotas[w] <= 1.000.000) }
 * fun resolver(cotas[0..N) de ent, D : ent) dev apta : bool
 * Q = { apta = !(Existe ini,fin : 0 <= ini < fin < N : tramoCreciente(cotas, ini, fin)) || ParaTodo i,j : 0 <= i < j < N & tramoCreciente(cotas, i, j) : cotas[j] - cotas[i] <= D }
 *
 * donde se utiliza el siguiente predicado auxiliar:
 * - tramoCreciente(v, ini, fin) = (ParaTodo w : ini <= w < fin : v[w] < v[w + 1])
 *
 * COSTE
 * -----
 * El coste asintótico en tiempo de la función es del orden O(N), donde N es el número de cotas de la excursión. Este coste lineal se debe
 * a que recorremos el vector de cotas pasando 1 vez por cada elemento y realizando operaciones de coste O(1) sobre cada uno de ellos.
 *
 * NOTA: El bucle interior no supone un aumento del coste, dado que aumenta el contador general que recorre el array y por lo tanto al ir
 * avanzando disminuye el número de vueltas restantes del bucle exterior.
 * En el caso mejor el coste disminuye puesto que no es necesario recorrer todo el vector (el caso mejor será aquel en el que v[1] - v[0] > D
 * por lo que nos saldremos directamente del bucle)
 */

// función que resuelve el problema
bool resolver(vector<int> const& cotas, int D) {
    bool apta = true;
    int ini = 0, fin = 0;
    int cont = 0;
```

```

while (apta && cont < cotas.size() - 1) {
    if (cotas[cont] < cotas[cont + 1]) {
        ini = cont;
        fin = cont + 1;
        ++cont;
        while (cont < cotas.size() - 1 && cotas[cont] < cotas[cont + 1]) {
            ++fin;
            ++cont;
        }
        if (cotas[fin] - cotas[ini] > D)
            apta = false;
    }
    ++cont;
}

return apta;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int D; cin >> D;
    if (!std::cin)
        return false;
    int N; cin >> N;
    vector<int> cotas(N);
    for (int i = 0; i < N; ++i)
        cin >> cotas[i];

    bool sol = resolver(cotas, D);

    // escribir sol
    if (sol) cout << "APTA";
    else cout << "NO APTA";
    cout << '\n';

    return true;
}

```

Todos con la selección

Enunciado:

Si la selección nacional gana el próximo partido en Málaga, habrá ganado 6 partidos seguidos. Hacía bastante tiempo que no tenía una racha ganadora seguida tan larga. Nuestro periodista encargado de seguir el partido del próximo sábado quiere saber cual ha sido la racha ganadora más larga de la selección en todo la historia, para poder contarla durante el partido.

Para ello recopila los datos y le pide a un amigo informático que le ayude a analizarlos con un programa. Deben obtener el máximo número de partidos seguidos que ha conseguido ganar la selección, si ha ocurrido varias veces que se ganasen este número de partidos, y hace cuantos partidos que finalizó la última racha.



Requisitos de implementación.

Implementar una función que reciba en un vector los datos, y devuelva la información pedida en el problema.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba tiene dos líneas. En la primera se indica el número de partidos jugados por la selección. En la segunda se indica la diferencia de goles entre los dos equipos. Un valor positivo indica que la selección ganó el partido, un valor cero indica que empató y un valor negativo que perdió.

Salida

Para cada caso de prueba se escribe en una línea el número máximo de partidos seguidos ganados, el número de veces que se ha ganado este número de partidos seguidos y el número de partidos jugados desde que finalizó la última racha ganadora.

Entrada de ejemplo

```
10
2 0 -3 1 1 0 2 1 -1 2
9
-1 3 1 2 0 1 -2 4 3
10
1 1 3 0 1 -1 4 3 1 2
3
-1 0 -1
```

Salida de ejemplo

```
2 2 2
3 1 5
4 1 0
0 0 3
```

Autor: Isabel Pita.

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *  ACLARACIONES:
 *  N = resultados.size() (= número de partidos jugados por la selección)
 *
 *  ESPECIFICACIÓN
 *  -----
 *  P = { 0 < N < 600.000 }
 *  fun resolver(resultados[0..N] de ent) dev maxRacha : ent, cuantasRachas : ent, partidosUltRacha : ent
 *  Q = { (maxRacha = max ini,fin : 0 <= ini <= fin < N /\ rachaGanadora(resultados, ini, fin) : fin - ini) /\ 
 *        (cuantasRachas = #ini,fin : 0 <= ini <= fin < N /\ rachaGanadora(resultados, ini, fin) : fin - ini = maxRacha) /\ 
 *        (partidosUltRacha = N - fin - 1 : ultimaRacha(resultados, N, ini, fin) } 
 *
 *  donde se utilizan los siguientes predicados auxiliares:
 *  · rachaGanadora(v, i, j) = (ParaTodo w : i <= w <= j : v[w] > 0)
 *  · ultimaRacha(v, n, ini, fin) = (rachaGanadora(v, ini, fin) /\ (ParaTodo i,j : 0 <= i <= j < N /\ rachaGanadora(v, i, j) : ini > i))
 *
 *  COSTE
 *  -----
 *  El coste asintótico en tiempo de la función es del orden O(N), donde N es el número de partidos
 *  jugador por la selección. Esto es porque se recorre el vector entero pasando 1 vez por cada uno de
 *  sus elementos y realizando operaciones de coste constante O(1) sobre ellos.
 *
 *  NOTA: El bucle interior no supone un aumento en el coste asintótico del programa (aunque sí dificulta la
 *  obtención del invariante del bucle) dado que este incrementa el contador y, por tanto, reduce el número
 *  de vueltas que dará el bucle exterior
 *
 */

struct solucion {
    int maxRacha;
    int cuantasRachas;
    int partidosUltRacha;
};

// función que resuelve el problema
solucion resolver(vector<int> const& resultados) {
    int iniMejorRacha = 0, finMejorRacha = -1;
    int cuantasRachas = 0;
    int iniAux, finAux;
    int cont = 0;
```

```

while (cont < resultados.size()) {

    if (resultados[cont] > 0) { // si ha ganado

        iniAux = cont;
        finAux = cont;
        ++cont;

        while (cont < resultados.size() && resultados[cont] > 0) { // estudiamos racha ganadora
            ++finAux;
            ++cont;
        }

        if (finAux - iniAux >= finMejorRacha - iniMejorRacha) { // si es racha ganadora

            if (finAux - iniAux > finMejorRacha - iniMejorRacha) // una mejor que la que teníamos
                cuantasRachas = 1;
            else // una igual a la que teníamos
                ++cuantasRachas;

            // actualizamos:
            iniMejorRacha = iniAux;
            finMejorRacha = finAux;

        }
    }
    ++cont;
}

int partidosUltRacha = resultados.size() - finMejorRacha - 1;
int maxRacha = finMejorRacha - iniMejorRacha + 1;

return { maxRacha, cuantasRachas, partidosUltRacha };
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int N; cin >> N;
    if (!std::cin)
        return false;
    vector<int> resultados;
    for (int i = 0; i < N; ++i)
        cin >> resultados[i];

    solucion sol = resolver(resultados);

    // escribir sol
    cout << sol.maxRacha << " " << sol.cuantasRachas << " " << sol.partidosUltRacha << '\n';

    return true;
}

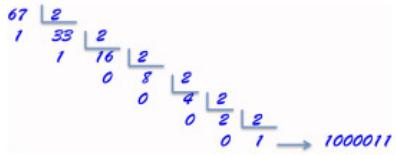
```

Transformar un número decimal a binario

Enunciado:

Con la revolución digital, el sistema de numeración en base dos o sistema de numeración binario adquiere una gran relevancia al ser el sistema utilizado por los ordenadores y las redes de comunicación. Todo informático necesita en algún momento obtener la representación binaria de un número.

En este problema desarrollaremos un conversor de números expresados en base 10 a números expresados en base 2.



Requisitos de implementación.

El conversor debe realizarse con una función que dado un número entero devuelva en una cadena de caracteres de tipo `std::string` su representación en binario.

Se utilizará una función `resuelveCaso` para leer el dato de entrada, llamar al conversor y escribir la cadena de salida.

Entrada

La entrada comienza con una línea en que se indica el número de casos de prueba. Cada caso consiste en un número entero positivo n ($0 \leq N \leq 2^{31} - 1$).

Salida

Para cada caso de prueba se muestra en una línea la representación en binario del número.

Entrada de ejemplo

```

6
3
8
1
24
156
345
  
```

Salida de ejemplo

```

11
1000
1
11000
10011100
101011001
  
```

Solución:

```

// // Beatriz Espinar Aragón
// // F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <string>
using namespace std;

/*
 *  ESPECIFICACIÓN FORMAL (binario)
 *
 *  -----
 *  P = { 0 <= n <= 2^31 - 1 /\ decimal(n) }
 *  fun binario(n : ent) dev s : string
 *  Q = { s = to_string(binario(n)) }
 *
 *  donde decimal(n) indica que el número n está expresado en base decimal, to_string(k) devuelve el número k
 *  expresado como cadena de caracteres, y se utiliza el siguiente predicado auxiliar:
 *  . binario(n) devuelve un n' que cumple: ((SUM i : 0 <= i < (max w : n' / 10^w != 0 : w) : ((n' / 10^i) % 10) * 2^i) = n)
 *
 *  RECURRENCIA (binario)
 *
 *  -----
 *  T(n) = { log n           si n = {0, 1}
 *          { T(n / 2) + log n   si n > 1
 *  */
  
```

```

/*
 * COSTE (binario)
 *
 * -----
 * Si aplicamos el Teorema de la División sobre la función de recurrencia (donde a = 1, b = 2, k = 0), obtenemos que
 * el coste asintótico en tiempo de la función recursiva es del orden O(log n), donde n es el número que queremos
 * transformar a binario
 *
 * NOTA: El coste de concatenar strings es lineal con respecto a la longitud de los strings, lo cual es logarítmico con
 * respecto al número n que estamos convirtiendo a decimal.
 *
 *
 * ESPECIFICACIÓN FORMAL (binario_final)
 *
 * -----
 * P = { 0 <= n <= 2^31 - 1 /\ decimal(n) /\ ¿(bin = parte que llevamos por ahora pasada a binaria)?? }
 * fun binario_final(n : ent, string bin) dev s : string
 * Q = { s = to_string(binario(n)) }
 *
 * donde decimal(n) indica que el número n está expresado en base decimal, to_string(k) devuelve el número k
 * expresado como cadena de caracteres, y se utiliza el siguiente predicado auxiliar:
 * . binario(n) devuelve un n' que cumple: ((SUM i : 0 <= i < (max w : n' / 10^w != 0 : w) : ((n' / 10^i) % 10) * 2^i) = n)
 */

* RECURRENCIA (binario)
*
* -----
* T(n) = { log(n)           si n = {0, 1}
*          { T(n / 2) + log(n)   si n > 1
*
* COSTE (binario)
* -----
* Si aplicamos el Teorema de la División sobre la función de recurrencia (donde a = 1, b = 2, k = 0), obtenemos que
* el coste asintótico en tiempo de la función recursiva es del orden O(log n), donde n es el número que queremos
* transformar a binario
*
* NOTA: El coste de concatenar strings es lineal con respecto a la longitud de los strings, lo cual es logarítmico con
* respecto al número n que estamos convirtiendo a decimal.
*/
*/

// función recursiva final
string binario_final(int n, string bin) {

    // CASO BASE:
    if (n == 0)
        return "0";
    if (n == 1)
        return "1" + bin;

    // CASO RECURSIVO:
    else
        return binario_final(n / 2, to_string(n % 2) + bin);
}

// la llamada sería binario_final(n, "");

// función recursiva
string binario(int n) {

    // CASO BASE:
    if (n == 0)
        return "0";
    if (n == 1)
        return "1";

    // CASO RECURSIVO:
    else
        return binario(n / 2) + to_string(n % 2);
}

// función que resuelve el problema
string resolver(int n) {
    return binario(n);
}

```

```

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;

    string sol = resolver(n);

    // escribir sol
    cout << sol << '\n';
}

```

Números interesantes

Enunciado:

- 2. (3.5 puntos)** Un numero natural se dice *interesante* cuando no contiene ningún cero, y cuando cada dígito divide a la suma de todos los dígitos que lo preceden, y también divide a la suma de todos los dígitos que lo suceden. Debe diseñarse e implementarse un algoritmo recursivo que, dado un número natural n , devuelva **true** si n es *interesante*, y **false** en caso contrario. Debe determinarse, asimismo, justificadamente el orden de complejidad del algoritmo.

La implementación deberá ir acompañada de un programa de prueba, que lea desde la entrada estandar casos de prueba, los ejecute, e imprima por la salida estándar el resultado. La entrada comenzará con el número de casos de prueba. Cada caso consistirá en una línea con el número n a analizar. El programa escribirá **SI** si el número n es *interesante*, o **NO** en otro caso.

A continuación se muestra un ejemplo de entrada / salida:

Entrada	Salida
3	
621348	SI
621356	NO
846213	SI

Solución:

```

// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
using namespace std;

/*
 * ESPECIFICACIÓN FORMAL
 *
 * P = { n >= 0 && sumDch >= 0 }
 * fun esInteresante(n : nat, sumDch : nat) dev <interesante : bool, sumIzq : nat>
 * Q = { interesante = noHayCeros(n) && (ParaTodo i : 0 <= i <= numDigitos(n) : suma(n, 0, i) % ((n / 10^i) % 10) = 0 /\ suma(n, i, numDigitos(n)) % ((n / 10^i) % 10) = 0 ) }
 *
 * donde se utilizan los siguientes predicados auxiliares:
 * - noHayCeros(n) = (ParaTodo i : 0 <= i <= numDigitos(n) : (n / 10^i) % 10 != 0)
 * - numDigitos(n) = (max w : n / 10^w != 0 : w)
 * - suma(n, i, j) = (SUM w : i <= w < j : (n / 10^w) % 10)
 *
 * RECURRENCIA
 *
 * -----
 * T(n) = { k_0           si n < 10 \vee n % 10 = 0
 *          { T(n / 10) + k_1   eoc
 *
 * COSTE
 *
 * -----
 * Aplicando el Teorema de la División a partir de la función de recurrencia (donde a = 1, b = 10, k = 0) obtenemos
 * que el coste asintótico en tiempo de la función recursiva es del orden O(log n), donde n es el valor de entrada.
 *
 * NOTA: El coste es del orden O(m) (lineal) donde m es el número de dígitos del número n de entrada.
 */

```

```

// función recursiva
pair<bool, long long int> esInteresante(long long int n, long long int sumDch) {
    // CASO BASE:
    if (n % 10 == 0) // si el número tiene un dígito 0 ya sabemos que no es interesante
        return { false, -1 }; // y nos da igual la suma de los dígitos anteriores
    else if (n < 10) // sólo 1 dígito
        return { sumDch % n == 0, n };

    // CASO RECURSIVO:
    else {
        long long int ultDigito = n % 10;
        // llamada recursiva:
        pair<bool, long long int> interIzq = esInteresante(n / 10, sumDch + ultDigito);
        /* es interesante si los dígitos anteriores cumplen las condiciones, si la suma de los anteriores es divisible por el
           dígito actual, y si la suma de los siguientes también lo es */
        bool interesante = interIzq.first && (interIzq.second % ultDigito == 0) && (sumDch % ultDigito == 0);
        return { interesante, interIzq.second + ultDigito };
    }
}

// función que resuelve el problema
bool resolver(long long int n) {
    return esInteresante(n, 0).first;
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    long long int n; cin >> n;

    bool sol = resolver(n);

    // escribir sol
    if (sol) cout << "SI";
    else cout << "NO";
    cout << '\n';
}

```

Elementos bien situados

Enunciado:

Descripción

En un vector de enteros **ordenado y sin repeticiones** `vector<int>` diremos que el elemento en posición i está *bien situado* si $v[i] == i$. Implementa una función `elemento_situado` con firma

```
bool elemento_situado(const vector<int>& v, int ini, int fin)
```

que detecta si el vector ordenado y sin repeticiones v contiene algún elemento bien situado. La solución iterativa natural tiene coste $O(n)$, pero debéis desarrollar una solución siguiendo la técnica de **divide y vencerás** con un **coste inferior**.

Utiliza la función `elemento_situado` para resolver los distintos casos de prueba que se leen por la entrada estándar.

Entrada

La entrada comenzará con una línea conteniendo un número natural N que indica la cantidad de vectores en los que buscar elementos bien situados. A continuación le siguen N líneas, cada una con una secuencia de números enteros que forman el vector. El primer número de la secuencia indica el número de elementos que tiene el vector. Ten en cuenta que el vector puede constar de 0 elementos.

Salida

Por cada vector se mostrará una línea con la palabra **SI** en caso de que el vector contenga algún elemento bien situado, y **NO** en otro caso.

Ejemplo de entrada

```
4
6 -8 -5 0 2 3 5
1 0
5 5 6 7 8 9
4 -4 -3 -2 -1
```

Ejemplo de salida

```
SI
SI
NO
NO
```

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *      ACLARACIONES:
 *      • N = v.size()
 *
 *      ESPECIFICACIÓN FORMAL
 *      -----
 *      P = { N >= 0 /\ 0 <= ini < fin <= N /\ ordenado_sin_repeticiones(v, N) }
 *      fun elemento_situado(v[0..N] de ent, ini : ent, fin : ent) dev sol : bool
 *      Q = { sol = (Existe i : ini <= i < fin : v[i] = i) }
 *
 *      donde se utiliza el siguiente predicado auxiliar:
 *      • ordenado_sin_repeticiones(v, n) = (ParaTodo w : 0 <= w < N - 1 : v[w] < v[w + 1])
 *
 *      RECURRENCIA
 *      -----
 *      T(N) = { k_0           si N = 0
 *              { T(N / 2) + k_1   si N > 0    (en el caso mejor sería sólo k_2, si v[m] = m)
 *
 *      COSTE
 *      -----
 *      Si aplicamos el Teorema de la División a partir de la función de recurrencia (donde a = 1, b = 2, k = 0)
 *      obtenemos que el coste asintótico en tiempo de la función recursiva es del orden O(log N)
 *
 */

// función recursiva
bool elemento_situado(vector<int> const& v, int ini, int fin) {
    // CASO BASE:
    if (ini == fin) // vector vacío
        return false;

    // CASO RECURSIVO:
    else {
```

```

        int m = (ini + fin) / 2;
        if (v[m] == m)
            return true;
        else if (v[m] > m)
            return elemento_situado(v, ini, m);
        else
            return elemento_situado(v, m + 1, fin);
    }
}

// función que resuelve el problema
bool resolver(vector<int> const& v) {
    return elemento_situado(v, 0, v.size());
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
void resuelveCaso() {

    // leer los datos de la entrada
    int N; cin >> N;
    vector<int> v(N);
    for (int i = 0; i < N; ++i)
        cin >> v[i];

    bool sol = resolver(v);

    // escribir sol
    if (sol) cout << "SI";
    else cout << "NO";
    cout << '\n';
}

```

Impares borrados

Enunciado:

Implementa un algoritmo recursivo eficiente que dado un vector de números enteros positivos y consecutivos del que se han eliminado todos los elementos impares menos uno, encuentre cuál es ese número impar que ha quedado en el vector. Plantea la recurrencia correspondiente al coste de este algoritmo e indica a qué orden de complejidad pertenece la solución a la misma.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de 2 líneas. En la primera se indica el número de elementos del vector una vez eliminados los elementos impares. En la siguiente se indican los elementos del vector. El final de los casos se marca con el valor cero. El vector tiene al menos un elemento. Se garantiza que siempre existe un elemento impar en el vector.

Salida

Para cada caso de prueba se escribe en una línea el elemento impar del vector.

Entrada de ejemplo

```

4
6 8 9 10
4
1 2 4 6
4
4 5 6 8
4
4 6 8 9
5
1 2 4 6 8
5
2 3 4 6 8

```

```
1
3
2
5 6
2
6 7
0
```

Salida de ejemplo

```
9
1
5
9
1
3
3
5
7
```

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <iomanip>
#include <fstream>
#include <vector>
using namespace std;

/*
 *   ACLARACIONES:
 *   - N = v.size()
 *
 *   ESPECIFICACIÓN FORMAL
 *   -----
 *   P = { N > 0 /\ 0 <= ini < fin <= N /\ positivos(v, N) /\ todos_pares_un_impar(v, N) /\ consecutivos(v, N) }
 *   fun numero_impar(v[0..N] de ent, ini : ent, fin : ent) dev impar : ent
 *   Q = { (impar % 2 = 1) /\ pertenece(v, N, impar) }
 *
 *   donde se utilizan los siguientes predicados auxiliares:
 *   - positivos(v, n) = (ParaTodo w : 0 <= w < n : v[w] > 0)
 *   - todos_pares_un_impar(v, n) = ((Existe! i : 0 <= i < n : v[i] % 2 = 1) /\ (ParaTodo w : 0 <= w < n /\ w != i : v[w] % 2 = 0))
 *   - consecutivos(v, n) = (ParaTodo w : 0 <= w < n - 1 : (v[w + 1] - v[w] = 2) \vee (v[w + 1] - v[w] = 1))
 *   - pertenece(v, n, k) = (Existe w : 0 <= w < n : v[w] = k)
 *
 *   RECURRENCIA
 *   -----
 *   T(N) = { k_0           si N = 1
 *           { T(N / 2) + k_1   si N > 1   (en el caso mejor sería constante si v[m] es el impar que buscamos)
 *
 *   COSTE
 *   -----
 *   Si aplicamos el Teorema de la División a partir de la función de recurrencia (donde a = 1, b = 2, k = 0) obtenemos que
 *   el coste asintótico en tiempo de la función recursiva es del orden O(log N). (Esto tiene sentido dado que hacemos una
 *   única llamada recursiva que reduce el tamaño del problema a la mitad)
 *
 */

// función recursiva
int numero_impar(vector<int> const& v, int ini, int fin) {
    // CASO BASE:
    if (ini == fin - 1) // si sólo hay 1 elemento debe ser impar
        return v[ini];

    // CASO RECURSIVO:
    else {
```

```

        int m = (ini + fin) / 2;
        if (v[m] % 2 == 1) // si es el impar que buscábamos
            return v[m];
        else if (v[m] - v[ini] < 2 * (m - ini))
            return numero_impar(v, ini, m);
        else
            return numero_impar(v, m + 1, fin);
    }
}

// función que resuelve el problema
int resolver(vector<int> const& v) {
    return numero_impar(v, 0, v.size());
}

// Resuelve un caso de prueba, leyendo de la entrada la
// configuración, y escribiendo la respuesta
bool resuelveCaso() {

    // leer los datos de la entrada
    int n; cin >> n;
    if (n == 0)
        return false;
    vector<int> v(n);
    for (int i = 0; i < n; ++i)
        cin >> v[i];

    int sol = resolver(v);

    // escribir sol
    cout << sol << '\n';
    return true;
}

```

Papa Noel reparte juguetes

Enunciado:

Cada año son más los niños que le piden regalos a Papa Noel. Esta noche debe repartir los regalos y todavía no tiene preparado lo que le dará a cada niño. Para poder llegar a tiempo los elfos han diseñado un programa informático, que asignará a cada niño 2 juguetes entre todos los disponibles. Como quieren que los niños queden contentos han elaborado una lista con la satisfacción que le produce a cada niño cada uno de los juguetes que tienen en la fábrica. El objetivo es que todos los niños puedan llegar a un cierto grado de satisfacción, aunque no reciban los juguetes que más les gustan. Papa Noel también ha pedido que los juguetes sean variados, para que ningún niño reciba dos cosas demasiado parecidas, para ello los elfos han clasificado los juguetes por tipos y no entregarán a un niño dos juguetes del mismo tipo.

El jefe elfo de informática ha puesto a su equipo a trabajar en un programa que obtenga todas las posibles asignaciones de juguetes que cumplen los requisitos pedidos por Papa Noel. Para evitar las asignaciones repetidas (por ejemplo dar a un niño el juguete A y el juguete B es la misma asignación que darle el juguete B y el juguete A) los dos juguetes de cada niño se obtendrán ordenados, el primer juguete será el de menor identificador y el segundo el de mayor identificador.



Requisitos de implementación.

El problema se debe implementar empleando la técnica de vuelta atrás.

En la plantilla propuesta, la solución se almacena en un vector donde las componentes pares tienen el primer juguete asignado a un niño y las componentes impares el segundo juguete. En las etapas pares se asigna el primer juguete de un niño y en las etapas impares el segundo juguete.

Entrada

La entrada consta de una serie de casos de prueba. Cada caso de prueba consta de $n+3$ líneas. En la primera se indica el número juguetes diferentes que se fabrican, m , el número de niños a los que se les reparten juguetes, n , y la mínima satisfacción pedida para los niños. En la siguiente línea se indica el número de unidades disponibles de cada juguete. En la tercera línea se indica el tipo que tiene cada juguete y en las n últimas líneas la satisfacción de cada uno de los n niños con cada uno de los m juguetes. Después de cada caso hay una línea en blanco para facilitar la identificación de los casos en el ejemplo.

El número de juguetes diferentes, el número de niños y la satisfacción mínima son enteros mayores que cero. El número de unidades disponibles es un entero mayor o igual que cero. El tipo de cada juguete es una cadena de caracteres, y la satisfacción es un número entero que puede ser negativo si el niño aborrece el juguete.

Salida

Para cada caso de prueba se escriben todas las posibles asignaciones que se pueden realizar, una en cada línea. Si no existe ninguna asignación posible se escribirá SIN SOLUCION. Después de cada caso se escribe una línea en blanco.

Entrada de ejemplo

```
4 3 10
2 1 1 2
Tipo1 Tipo1 Tipo2 Tipo3
8 9 3 1
6 4 5 3
2 2 9 9

4 3 8
2 1 1 2
Tipo1 Tipo1 Tipo2 Tipo3
8 9 3 1
6 4 5 3
2 2 9 9

4 2 10
1 1 1 1
Tipo1 Tipo1 Tipo1 Tipo2
10 12 12 15
9 10 20 7

3 5 10
4 3 5
Tipo1 Tipo1 Tipo2
5 7 4
8 6 5
4 8 2
9 3 4
8 5 5
```

Salida de ejemplo

```
1 3 0 2 0 3

0 2 0 3 1 3
0 3 0 2 1 3
0 3 0 3 1 2
0 3 1 2 0 3
1 2 0 3 0 3
1 3 0 2 0 3
1 3 0 3 0 2

SIN SOLUCION

1 2 0 2 1 2 0 2 0 2
1 2 0 2 1 2 0 2 1 2
1 2 1 2 1 2 0 2 0 2
```

Autor: Isabel Pita

Solución:

```

// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <fstream>
#include <vector>
#include <string>

using namespace std;

using matriz = vector<vector<int>>; // Matriz de enteros

/*
 * Añade las funciones auxiliares que necesites.
 * e.g. para escribir la solución, comprobar si es valida, etc
 */

/* comprueba que hemos repartido todos los regalos, que quedan existencias del último regalo que estamos vendiendo (el resto se suponen comprobados), que el niño al que estamos repartiendo el regalo no tenía un regalo del tipo del regalo que estamos vendiendo (el resto se suponen ya comprobados) y que la satisfacción del último niño es suficiente (el resto se suponen ya comprobados) */
bool es_satisfactoria(const int k, const int n, matriz const& satisfacciones, vector<int> const& existencias, vector<string> const& tipos, const int sat_min) {
    vector<int> const& sol = sol;
    vector<int> const& cuantos_vendidos = cuantos_vendidos;
    vector<string> const& tipos_ninjo = tipos_ninjo;
    const int sat_min = sat_min;
    return k == n - 1 && cuantos_vendidos[sol[k]] < existencias[sol[k]] && tipos[sol[k]] != tipos_ninjo[k / 2] && satisfacciones[k / 2][sol[k - 1]] + satisfacciones[k / 2][sol[k]] >= sat_min;
}

/* comprueba que todavía quedan regalos por repartir, que quedan existencias del último regalo que estamos vendiendo (el resto se suponen comprobados), que el niño al que estamos repartiendo el regalo no tenía un regalo del tipo del regalo que estamos vendiendo (el resto se suponen ya comprobados) y que la satisfacción del niño es suficiente (el resto se suponen ya comprobados) */
bool es_completamente(int k, const int n, matriz const& satisfacciones, vector<int> const& existencias, vector<string> const& tipos, vector<int> const& sol, vector<int> const& cuantos_vendidos, vector<string> const& tipos_ninjo, const int sat_min) {
    bool completable;
    completable = k < n - 1 && cuantos_vendidos[sol[k]] < existencias[sol[k]];
    if ((k % 2 == 1) // si es el primer regalo no hay ningún tipo que comprobar ni conocemos todavía la satisfacción del niño
        completable = completable && tipos[sol[k]] != tipos_ninjo[k / 2] && satisfacciones[k / 2][sol[k - 1]] + satisfacciones[k / 2][sol[k]] >= sat_min;
    return completable;
}

/* comprueba si con la satisfacción que tiene el niño actual con su primer regalo tiene opciones de conseguir la satisfacción mínima requerida. Si no, podemos */
bool es_prometedora(int k, vector<int> const& max_sat, int sat_min, int sat_actual) {
    return sat_actual + max_sat[k / 2] >= sat_min;
}

// muestra la distribución por pantalla
void procesar_solucion(vector<int> const& sol, int n) {
    for (int i = 0; i < n; ++i)
        cout << sol[i] << " ";
    cout << '\n';
}

/*
 * ESPACIO DE BÚSQUEDA
 * -----
 * El espacio de búsqueda es un árbol en el que la solución es una tupla ( $X_0, \dots, X_{n-1}$ ), donde  $n = 2 * \text{num\_ninos}$ 
 * y ParaTodo  $k : 0 \leq k < n : (k \% 2 = 0 \rightarrow X_k = \text{sol}[k] = \text{primer regalo que recibe el niño } k / 2) \wedge (k \% 2 = 1 \rightarrow X_k = \text{sol}[k] = \text{segundo regalo que recibe el niño } k / 2) \wedge 0 \leq X_k \leq \text{num\_juegos}$ 
 * MARCADORES
 * -----
 * Los marcadores empleados para mejorar la eficiencia de las funciones es_solución() y es_completamente() son los siguientes:
 * - cuantos_vendidos: ParaTodo  $i : 0 \leq i < \text{num\_juegos} : (\text{cuantos\_vendidos}[i] = \text{cuántas uds del regalo } i \text{ se han vendido}) \wedge (0 \leq \text{cuantos\_vendidos}[i] < \text{existencias}[i])$ 
 * - tipos_ninjo: ParaTodo  $i : 0 \leq i < \text{num\_ninos} : \text{tipos\_ninjo}[i] = \text{tipo del primer regalo que ha recibido el niño } i (" si no ha recibido todavía ninguno)$ 
 * PODA
 * -----
 * Aunque no supone un gran cambio en el coste asintótico de la función, se utiliza una función de poda sencilla que consiste en lo siguiente:
 * Se emplea un vector max_sat que almacena la satisfacción máxima que puede llegar a tener un niño con cualquier regalo. Así,
 * si el niño ya ha recibido un regalo, y la satisfacción de este sumada a esa satisfacción máxima no alcanza el mínimo de satisfacción requerido, podamos el árbol. No tiene sentido buscar un segundo regalo para ese niño si ya con el que tiene es imposible que llegue a tener la mínima satisfacción ni siquiera en el mejor de los casos.
 */

// crea la función a la que le pases la estructura para tu solución y los marcadores que uses
// Esta función saca por pantalla las soluciones
void papaNoelRecursoivo(bool& hay_solucion, matriz const& satisfacciones, vector<int> const& existencias, vector<string> const& tipos, vector<int> const& max_sat, const int sat_min, const int k, const int num_juguetes, const int p, const int n, vector<int>& sol, vector<int>& cuantos_vendidos, vector<string>& tipos_ninjo) {
    if ((k % 2 == 0) { // si es el primer regalo
        for (int i = 0; i < num_juguetes; ++i) {
            sol[k] = i;
            if (es_completamente(k, n, satisfacciones, existencias, tipos, sol, cuantos_vendidos, tipos_ninjo, sat_min) && es_prometedora(k, max_sat, sat_min, satisfacciones[k / 2][sol[k]])) {
                // marcamos:
                ++cuantos_vendidos[i];
                tipos_ninjo[k / 2] = tipos[i];
                // vuelta atrás:
                papaNoelRecursoivo(hay_solucion, satisfacciones, existencias, tipos, max_sat, sat_min, k + 1, num_juguetes, i + 1, n, sol, cuantos_vendidos, tipos_ninjo);
                // desmarcamos:
                --cuantos_vendidos[i];
                tipos_ninjo[k / 2] = "";
            }
        }
        // no hace falta comprobar si es solución
    }
}

```

```

    else { // si es el segundo regalo no miramos opciones repetidas (0,3 = 3,0)
        for (int i = p; i < num_juguetes; ++i) {
            sol[k] = i;
            if (es_solucion(k, n, satisfacciones, existencias, tipos, sol, cuantos_vendidos, tipos_ninjo, sat_min)) {
                hay_solucion = true; // esto realmente sólo hace falta hacerlo 1 vez
                procesar_solucion(sol, n);
            }
            else if (es_completable(k, n, satisfacciones, existencias, tipos, sol, cuantos_vendidos, tipos_ninjo, sat_min)) {
                // marcamos:
                ++cuantos_vendidos[i];
                // vuelta atrás:
                papaNoelRecurso(hay_solucion, satisfacciones, existencias, tipos, max_sat, sat_min, k + 1, num_juguetes, p, n, sol, cuantos_vendidos, tipos_ninjo);
                // desmarcamos:
                --cuantos_vendidos[i];
            }
        }
    }

//Devuelve true si hay una solución y false si no hay ninguna
bool papaNoel(vector<int> const& existencias, vector<string> const& tipos, matriz const& satisfacciones, const int num_juguetes, const int num_ninjos, const int sat_min, vector<int> const& max_sat) {
    int k = 0, p = 0, n = 2 * num_ninjos;
    bool hay_solucion = false;
    vector<int> sol(n);
    // marcadores:
    vector<int> cuantos_vendidos(num_juguetes, 0);
    vector<string> tipos_ninjo(num_ninjos, "");

    papaNoelRecurso(hay_solucion, satisfacciones, existencias, tipos, max_sat, sat_min, k, num_juguetes, p, n, sol, cuantos_vendidos, tipos_ninjo);

    return hay_solucion;
}

bool resuelveCaso() {

    // lectura de los datos de la entrada
    int jTotales, nNinos, satMin; // juguetes, niños y satisfacción mínima
    cin >> jTotales;
    if (!cin) // fin de la entrada
        return false;
    cin >> nNinos >> satMin;

    vector<int> existencias(jTotales); // existencias de juguetes
    vector<string> tipo(jTotales); // tipos de Juguetes
    matriz satisfaccion(nNinos, vector <int>(jTotales)); // satisfacción por niño y juguete
    vector<int> max_sat(nNinos);

    for (int i = 0; i < jTotales; i++)
        cin >> existencias[i];

    for (int i = 0; i < jTotales; i++)
        cin >> tipo[i];

    for (int i = 0; i < satisfaccion.size(); i++) {
        for (int j = 0; j < satisfaccion[0].size(); j++) {
            cin >> satisfaccion[i][j];
            if (j == 0 || max_sat[i] < satisfaccion[i][j]) // vamos calculando la máx satisfacción para cada niño
                max_sat[i] = satisfaccion[i][j];
        }
    }

    //Llama a la función
    bool haySol = papaNoel(existencias, tipo, satisfaccion, jTotales, nNinos, satMin, max_sat);

    //si no hay solución saca SIN SOLUCION
    if (!haySol)
        cout << "SIN SOLUCION" << endl;
    cout << endl;

    return true;
}

```

Concierto benéfico

Enunciado:

El 13 de Julio de 2019 se va a celebrar un concierto benéfico de rock como el que tuvo lugar hace 34 años en Wembley. Los artistas participantes ya están confirmados y solamente falta decidir el orden de actuación de los mismos. Los promotores del concierto han realizado una estimación de la cantidad de donaciones que se pueden recibir durante la actuación de cada uno de los n artistas dependiendo del momento 0 a $n - 1$ en el que actúan. También disponen de una tabla de "vetos" en la que cada artista ha reflejado si admite tocar o no inmediatamente después de cada uno de los demás. Por ejemplo Queen no acepta tocar después de nadie mientras que U2 acepta tocar solamente después de Chimpanzee. Ayuda

a los promotores a determinar el orden en que han de tocar los artistas para obtener la máxima donación posible según la estimación realizada.

- Implementa un algoritmo de vuelta atrás que resuelva el problema. Explica claramente los marcadores que has utilizado.
- Plantea dos posibles funciones de poda de optimalidad, razona sobre cual de ellas es mejor e impleméntala en tu algoritmo.

Entrada

La entrada comienza con una línea que contiene el número de casos de prueba. Cada caso de prueba contendrá el valor del número de artistas n . A continuación figuran las estimaciones de las donaciones: una fila para cada artista. Después los vetos de los artistas: una fila para cada artista i indicando si admite (1) o no (0) tocar después del artista j (habrá un 0 en la posición i).

Salida

Por cada caso de prueba el programa escribirá una línea con la donación máxima estimada (suma de las donaciones obtenidas por cada artista en el momento que le corresponde tocar). En caso de que no sea posible satisfacer los vetos se escribirá NEGOCIA CON LOS ARTISTAS.

Entrada de ejemplo

```
2
3
10 20 30
140 20 10
160 10 20
0 1 1
0 0 1
0 0 0
3
10 20 30
140 20 10
160 10 20
0 0 1
0 0 1
0 0 0
```

Salida de ejemplo

```
210
NEGOCIA CON LOS ARTISTAS
```

Solución:

```
// Beatriz Espinar Aragón
// F19

#include <iostream>
#include <fstream>
#include <algorithm>
#include <vector>
using namespace std;

using matriz = vector<vector<int>>

/*
 * Añade las funciones auxiliares que necesites.
 * e.g. para escribir la solución, comprobar si es valida, etc
 */
```

```

/* comprueba que el último artista que hemos asignado no tenía ya antes otra actuación
   (los anteriores se suponen ya comprobados) y que este no tiene vetado al artista que actúa antes que él */
bool es_valido(const int k, vector<int> const& sol, vector<bool> const& marcador, matriz const& vetos) {
    return !marcador[sol[k]] && (k == 0 || vetos[sol[k]][sol[k - 1]] == 1);
}

// calcula la donación total que tendremos tomando la última decisión
int actualizar(const int don_actual, matriz const& donaciones, vector<int> const& sol, const int k) {
    return don_actual + donaciones[sol[k]][k];
}

// vuelve a la donación que teníamos antes de tomar la última decisión
int desactualizar(const int don_actual, matriz const& donaciones, vector<int> const& sol, const int k) {
    return don_actual - donaciones[sol[k]][k];
}

// comprueba que ya hemos asignado todas las actuaciones
bool es_solucion(const int k, const int n) {
    return k == n - 1;
}

// comprueba si la nueva solución es mejor a la que llevábamos hasta ahora
bool es_mejor(const int don_actual, const int max_don) {
    return don_actual > max_don;
}

// comprueba que todavía quedan actuaciones sin asignar
bool es_completable(const int k, const int n) {
    // si es el primer artista nadie actúa antes que él
    return k < n - 1;
}

// es prometedora si la solución que llevamos hasta ahora más la máxima que podemos conseguir es mejor que la mejor guardada hasta ahora
bool es_prometedora(const bool solución, const int don_actual, const int max_don, const int max_don_restante) {
    return don_actual + max_don_restante > max_don;
}

/*
 * ESPACIO DE BÚSQUEDA
 * -----
 * El espacio de búsqueda es un árbol en el que la solución es una tupla  $(x_0, \dots, x_{n-1})$ , donde  $n = n^{\circ}$  de actuaciones ( $- n^{\circ}$  de artistas)
 * y ParaTodo  $k : 0 \leq k < n : (x_k = \text{sol}[k]) = \text{artista que actúa en el momento } k \wedge 0 \leq x_k \leq n$ 
 *
 * MARCADORES
 * -----
 * - marcador: ParaTodo  $i : 0 \leq i < n : \text{marcador}[i] = \text{el artista } i \text{ ya tiene una actuación asignada}$ 
 *
 * PODA
 * -----
 * Las dos posibles funciones de poda para optimizar el coste del algoritmo son:
 * 1) Llevar la máxima donación que puede recibir cualquier artista en cualquier actuación. Si llevamos una solución parcial, el mejor de los casos
 *    es que el resto de actuaciones recibieran esa máxima donación. Si ese valor es menor que el que llevamos hasta ahora, podamos.
 * 2) Llevar para cada actuación, la máxima donación que se puede recibir (la donación del artista que recibe más donación en esa actuación). Análoga a
 *    la anterior, si llevamos una solución parcial, el mejor de los casos es que las actuaciones que quedan recibieran su máxima donación. Si ese valor
 *    es menor que el que llevamos hasta ahora, podamos.
 * La función de poda más restrictiva y, por tanto, la mejor (quita más casos), es la segunda, que es la que se implementa en el algoritmo. Además,
 * se mejora la poda dando un valor inicial a la mejor solución. Se calcula la mínima donación (sea o no posible solución) que se puede obtener en cada
 * actuación, y el valor total será el valor inicial de max_don, en lugar de -1.
 */
}

// algoritmo vuelta atrás
void maxima_donación(bool& solución, const int n, matriz const& donaciones, matriz const& vetos, const int k, vector<int>& sol, int& don_actual, int& max_don, vector<bool>& marcador, vector<int> const& max_donaciones, int& max_don_restante) {
    for (int i = 0; i < n; ++i) {
        sol[k] = i;

        if (es_valido(k, sol, marcador, vetos)) {
            don_actual = actualizar(don_actual, donaciones, sol, k);
            max_don_restante -= max_donaciones[k];

            if (es_solucion(k, n)) {
                solución = true;
                if (es_mejor(don_actual, max_don))
                    max_don = don_actual;
            }
        }
    }
}

```

```

        } else if (es_completable(k, n) && es_prometedora(solucion, don_actual, max_don, max_don_restante)) {
            // marcamos:
            marcador[i] = true;
            // vuelta atrás:
            maxima_donacion(solucion, n, donaciones, vetos, k + 1, sol, don_actual, max_don, marcador, max_donaciones, max_don_restante);
            // desmarcamos:
            marcador[i] = false;
        }

        don_actual = desactualizar(don_actual, donaciones, sol, k);
        max_don_restante += max_donaciones[k];
    }
}

// función que resuelve el problema
int resolver(const int n, matriz const& donaciones, matriz const& vetos, vector<int> const& max_donaciones, int& max_don_restante, const int min_don) {

    // inicializamos parámetros y marcadores:
    int don_actual = 0, max_don = min_don;
    vector<int> sol(n);
    int k = 0;
    vector<bool> marcador(n, false);
    bool solucion = false;

    // llamada recursiva:
    maxima_donacion(solucion, n, donaciones, vetos, k, sol, don_actual, max_don, marcador, max_donaciones, max_don_restante);

    if (solucion) return max_don;
    else return -1;
}

```

```

void resuelveCaso() {

    int n; // n número de artistas
    cin >> n;

    matriz donaciones(n, vector<int>(n)); // matriz de donaciones
    matriz vetos(n, vector<int>(n)); // matriz de vetos

    // variables para implementar la poda
    vector<int> max_donaciones(n, -1), min_donaciones(n, -1);
    int max_don_restante = 0, min_don = 0;

    // leemos donaciones:
    for (int i = 0; i < n; i++) {
        for (int j = 0; j < n; ++j) {
            cin >> donaciones[i][j];
            // para la poda
            // máximas:
            if (max_donaciones[j] == -1 || max_donaciones[j] < donaciones[i][j]) {
                if (max_donaciones[j] != -1)
                    max_don_restante -= max_donaciones[j];
                max_donaciones[j] = donaciones[i][j];
                max_don_restante += max_donaciones[j];
            }
            // mínimas:
            if (min_donaciones[j] == -1 || min_donaciones[j] > donaciones[i][j]) {
                if (min_donaciones[j] != -1)
                    min_don -= min_donaciones[j];
                min_donaciones[j] = donaciones[i][j];
                min_don += min_donaciones[j];
            }
        }
    }

    // leemos vetos:
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; ++j)
            cin >> vetos[i][j];

    // Llamada recursiva.
    // Puedes usar un int para saber si es posible crear el concierto o no:
    int maxD = resolver(n, donaciones, vetos, max_donaciones, max_don_restante, min_don);

    if (maxD != -1)
        cout << maxD << endl;
    else
        cout << "NEGOCIA CON LOS ARTISTAS" << endl;
}

```